# Http基础

# 资源

## 资源定位

### URI

URI(Uniform Resource Identifier),统一资源定位符.

URIL,提供了一种定位因特网上任意资源的手段,但这些资源是可以通过各种不同的方案(比如HTTP, FTP, SMTP)来访问的,因此URL语法会随方案的不同而有所不同。

## URL通用组件

一般由三部组成: 方案(scheme).主机(host)和路径(path)

URI格式

①访问资源的命名机制

②存放资源的主机名

③资源自身的名称，由路径表示，着重强调于资源。

### 方案

方案: 访问服务器以获取资源时要使用哪种协议。

方案实际上是规定如何访向指定资源的主要标识符,它会告诉负责解析URL的应用是序应该使用什么协议,在我们这个简单的HTTP URL中所使用的方案就是http。

格式:字母符号+”:”+URL其他的部分

1、方案组件必须以一个字母符号开始,由第一个”:”符号将其余的URL的其他部分分隔开来。

2、不区分大小写。

### 主机和端口

主机组件标识了因特网上能够访问资源的宿主机器,可以用上述主机名,或者IP地址来表示主机名。

端口组件标识了服务器正在监听的网络端口。对下层使用了TCP协议的HTTP来说,默认端口号为80。

### 用户名和密码(HTTP认证)

更有趣的组件是用户和密码组件。很多服务器都要求输入用户名和密码才会允许用户访问数据.FTP服务器就是这样一个常见的实例。这里有几个例子:

图片包含 游戏机

描述已自动生成

第一个例子没有用户或密码组件,只有标准的方案、主机和路径。如果某应用程序使用的URL方案要求输入用户名和密码,比如FTP,但用户没有提供,它通常会插入一个默认的用户名和密码。比如,如果向浏览器提供一个FTP URL,但没有指定用户名和密码,它就会插入anonymous (匿名用户)作为你的用户名,并发送一个默认的密码(Internet Explorer会发送IEUser, Netscape Navigator则会发送mozilla).

### URL(地址定位)

URL是通过描述资源的位置来标识资源

源的

2、组成

①协议(或称为服务方式)

②存有该资源的主机IP地址(有时也包括端口号)

③主机资源的具体地址。如目录和文件名等

### URN统一资源名称(名称定位)

URI的第二种形式就是统一资源名(URN), URN是作为特定内容的唯一名称使用的,与目前的资源所在地无关。使用这些与位置无关的URN,就可以将资源四处搬移。通过URN,还可以用同一个名字通过多种网络访问协议来访问资源。

URN (本章稍后会介绍)则是通过名字来识别资源。

### URI、URL、URN的关系

图片包含 游戏机, ipod

描述已自动生成

例如:

URI统一资源标志符就是在某一规则下能把一个资源独一无二地标识出来。拿人做例子，假设这个世界上所有人的名字都不能重复，那么名字就是URI的一个实例，通过名字这个字符串就可以标识出唯一的一个人。现实当中名字当然是会重复的，所以身份证号才是URI，通过身份证号能让我们能且仅能确定一个人。

那统一资源定位符URL是什么呢。也拿人做例子然后跟HTTP的URL做类比，就可以有：

动物住址协议://地球/中国/浙江省/杭州市/西湖区/某大学/14号宿舍楼/525号寝/张三.人

可以看到，这个字符串同样标识出了唯一的一个人，起到了URI的作用，所以URL是URI子集。URL是以描述人的位置来唯一确定一个人的。在上文我们用身份证号也可以唯一确定一个人。对于这个在杭州的张三，我们也可以用：

身份证号：123456789

来标识他。

所以不论是用定位的方式还是用编号的方式，我们都可以唯一确定一个人，都是URl的一种实现，而URL就是用定位的方式实现的URI。

回到Web上，假设所有的Html文档都有唯一的编号，记作html:xxxxx，xxxxx是一串数字，即Html文档的身份证号码，这个能唯一标识一个Html文档，那么这个号码就是一个URI。而URL则通过描述是哪个主机上哪个路径上的文件来唯一确定一个资源，也就是定位的方式来实现的URI。

### 例如

<https://zhidao.baidu.com/question/1994186839382027827.html>

URL的第一部分(http)是URL方案(scheme),方案可以告知Web客户端怎

样访问资源,在这个例子中, URL说明要使用HTTP协议.

URL的第二部分zhidao.baidu.com)指的是服务器的位置。这部分告知

Web客户端资源住于何处。

URL的第三部分(/1994186839382027827.html)是资源路径,路径说明了请求的是

服务器上哪个特定的本地资源。

### java类应用

比如在JDK中sun公司提供的简易HttpServer实现中

public void handle（final HttpExchange exchange）throws Exception方法中，根据exchange对象可以拿到访问Http请求的URI对象，

ps：<http://127.0.0.1:8080/cmd_helloworld/?name=guowuxin>

此时URI uri = exchange.getRequestURI()；

通过uri可以拿到连接的各部分内容：

uri.getPath() --------------------> /cmd\_helloworld 注意有斜杠

uri.getQuery()----------------------> name=guowuxin

### 路径

路径是服务器定位资源时所需的信息，可以用字符”/”豆浆http url的路径划分成一些路径段(path segment),每个路径都有自己的参数param组件。

### 参数

负责解析URL的应用程序需要这些协议参数来访问资源。否则,另一端的服务器可能就不会为请求提供服务,或者更糟栏的是,提供错误的服务.比如,像FTP这样的协议,有两种传输模式,二进制和文本形式。你肯定不希望以文本形式来传送二进制图片,这样的话,二进制图片可能会变得一团糟。

为了向应用程序提供它们所需的输入参数,以便正确地与服务器进行交互, URL中有一个参数组件,这个组件就是URL中的名值对列表,由字符“:”其余部分(以及各名值对)分隔开来。它们为应用程”提供了访问资源所需的所有附加信息。

比如:

<http://www.joes-hardware.com/hammers;sale-false/index.html:graphics-true>

这个例子就有两个路径段, hammers和index.html. hammers路径段有参数sale,其值为false, index.html段有参数graphics,其值为true.

### 查询字符串(查询组件)

http://www.baidu.com?userName=1

这个URL的大部分都与我们见过的其他URL类似。只有问号(?)右边的内容是新出现的,这部分被称为查询(query)组件。URL的查询组件和标识网关资源的URL路径组件一起被发送给网关资源,基本上可以将网关当作访问其他应用程序的访问点。

格式：

1、名/值 对形式.

2、名值对之间使用&分割。

### 片段

## URL的组成

### 基础url

基础URL是作为相对URL的参考点使用的。可以来自以下几个不同的地方。转换处理的第一步就是找到基础URL.基础URL是作为相对URL的参考点使用的。可以来自以下几个不同的地方.

1、在資源中显式提供

有些资源会显式地指定基础URL.比如, HTML文档中可能会包含一个定义了基础URL的HTML标记<BASB>,通过它来转换那个HTML文档中的所有相对URL.

2、封装资源的基础URL

如果在一个没有显式指定基础URL的资源中发现了一个相对URL,如例2-1所斥,可以将它所属资源的URL作为基础(如例中所示)。

3、没有基础URL

在某些情况下,没有基础URL,这通常意味着你有一个相对URL,但有时可能只是一个不完整或损坏了的URL.

### 解析相对引用

要将相对URL转换为一个绝对URL,下一步要做的就是将相对URL和基础URL划分成组件段。

实际上,这样只是在解析URL,但这种做法会将其划分成一个个组件,因此通常会称作分解(decomposing) URL.只要将基础和相对URL划分成了组件。

HTTP方法

HTTP支持几种不同的请求命令,这些命令被称为HTTP方法(HTTP method)。每条HTTP请求报文都包含一个方法。这个方法会告诉服务器要执行什么动作(获取一个web页面、运行一个网关程序、删除一个文件等。

图片包含 游戏机, 截图

描述已自动生成

## 状态码

### hhttp保温2

# HTTP请求

## Content-Type

### Content-Type对get请求和post请求的影响

GET 请求不存在请求实体部分，键值对参数放置在 URL 尾部，浏览器把form数据转换成一个字串（name1=value1&name2=value2...），然后把这个字串追加到url后面，用?分割，加载这个新的url。因此请求头不需要设置 Content-Type 字段

非 ASCII 码会自动进行编码转换，例如发送请求：www.bilibili.com?hehe=你的我的

值得一提的是，GET 参数的编码方式是无法人为干涉的，这导致了不同浏览器有不同的编码方式，因此最稳妥的方案是人工预编码，人工解码，从而禁止浏览器编码的干涉

1、如何禁止解码

### POST 请求

Http Header里的Content-Type一般有这三种：

1.application/x-www-form-urlencoded：数据被编码为名称/值对。这是标准的编码格式。默认行为。会将表单内的数据转换拼接成 key-value 对（非 ASCII 码进行编码）

2.multipart/form-data(一般用来上传文件)： 数据被编码为一条消息，页上的每个控件对应消息中的一个部分

3.text/plain： 数据以纯文本形式(text/--json/xml/html)进行编码，其中不含任何控件或格式字符。postman软件里标的是RAW。（中文不进行编码）

### AJAX请求时最好要通过设置请求头来指定为application/x-www-form-urlencoded编码类型

### 表单上传文件时必须指定编码类型为"multipart/form-data

## application/x-www-form-urlencoded

### 编码原理

step 1 ：将保留字符转化为对应的 ASCⅡ 字节值 （byte value）;

step 2 ：将对应的对应的 ASCⅡ 字节值（十进制）转换成对应的十六进制值。

step 3 ：在对应十六进制值前面加上百分号 % 。

将需要转码的字符转为16进制，然后从右到左，取4位(不足4位直接处理)，每2位做一位，前面加上%，编码成%XY格式。

空格ASCII码是32，对应16进制是20，那么urlencode编码结果是:%20,但在新标准中空格对应的是+,见RFC-1738

中ASCII码是-10544，对应的16进制是D6D0，那么urlencode编码结果是:%D6%D0。

### 为什么要使用 URL编码 百分号编码

每一个 URL 都是单独的字符串。作为一个连续的字符串，URL 是不能存在空格的。所以在使用 URL 的时候需要使用其他的字符代替空格。

URL 中经常使用到的 ？ 和 # 等符号在 URL 中具有特殊的定义，在正常使用时，不能直接使用这些符号。所以我们需要使用其他字符代替这些符号。

根据 RFC 3986 协议定义，URL 对于在 URL 中用到的特殊字符和关键字，都需要使用特定的 URL 编码进行转化。

# JAVA HTTP

## URL类

### url类简介

在 java.net 包中包含专门用来处理 URL 的类 URL，可以获得 URL 的相关信息，例如 URL 的协议名和主机名等。下面分别对它的构造方法和常用方法进行介绍。

### 构造方法

|  |  |
| --- | --- |
| 构造方法 | 说明 |
| public URL (String spec) | 通过一个表示 URL 地址的字符串可以构造一个 URL 对象。 |
| public URL(URL context,String spec) | 使用基本地址和相对 URL 构造一个 URL 对象。 |
| public URL(String protocol,String host,String file) | 使用指定的协议、主机名和文件名创建一个 URL 对象。 |
| public URL(String protocol,String host,int port,String file) | 使用指定的协议、主机名、端口号和文件名创建一个 URL 对象。 |

2、URL(URL context,String spec)

URL url1 = new URL("http://www.baidu.com");

URL url2 = new URL(url1,"/index.html");

System.out.println("URL 为：" + url2.toString());

//URL 为：<http://www.baidu.com/index.html>

3、

URL url4 = **new** URL(**"https"**,**"www.baidu.com"**,**"/index.html"**);  
System.***out***.println(**"URL 为："** + url4.toString());  
*//URL 为：https://www.baidu.com/index.html*

4、

URL url5 = **new** URL(**"https"**,**"www.baidu.com"**,80,**"/index.html"**);  
System.***out***.println(**"URL 为："** + url5.toString());

### 属性

|  |  |
| --- | --- |
| 方法 | 说明 |
| public String getProtocol() | 获取该 URL 的协议名。 |
| public String getHost() | 获取该 URL 的主机名。 |
| public int getPort() | 获取该 URL 的端口号，如果没有设置端口，返回 -1。 |
| public String getFile() | 获取该 URL 的文件名。 |
| public String getRef() | 获取该 URL 在文件中的相对位置。 |
| public String getQuery() | 获取该 URL 的查询信息。 |
| public String getPath() | 获取该 URL 的路径。 |
| public String getAuthority() | 获取该 URL 的权限信息。 |
| public String getUserInfo() | 获得使用者的信息。 |
| public String getRef() | 获得该 URL 的锚点。 |

### 例如

|  |
| --- |
| URL url = **new** URL(**"https"**,**"www.baidu.com"**,**"/index/class?aaa=1"**); System.***out***.println(**"URL 为："** + url.toString()); System.***out***.println(**"协议为："** + url.getProtocol()); System.***out***.println(**"验证信息："** + url.getAuthority()); System.***out***.println(**"文件名及请求参数："** + url.getFile()); System.***out***.println(**"主机名："** + url.getHost()); System.***out***.println(**"路径："** + url.getPath()); System.***out***.println(**"端口："** + url.getPort()); System.***out***.println(**"默认端口："** + url.getDefaultPort()); System.***out***.println(**"请求参数："** + url.getQuery()); System.***out***.println(**"定位位置："** + url.getRef()); |

## Keep-alive

### Http中的Keep-Alive

HTTP 持久连接（HTTP persistent connection，也称作HTTP keep-alive或HTTP connection reuse，翻译过来可以是保持连接或者连接复用）是使用同一个TCP连接来发送和接收多个HTTP请求/应答，而不是为每一个新的请求/应答打开新的连接的方式。

HTTP协议采用“请求-应答”模式，当使用普通模式，即非KeepAlive模式时，每个请求/应答客户和服务器都要新建一个连接，完成 之后立即断开连接（HTTP协议为无连接的协议），每次请求都会经过三次握手四次挥手过程，效率较低；当使用Keep-Alive模式时，客户端到服务器端的连接不会断开，当出现对服务器的后继请求时，客户端就会复用已建立的连接。

图片包含 游戏机

描述已自动生成

在Http 1.0中，Keep-Alive是没有官方支持的，但是也有一些Server端支持，这个年代比较久远就不用考虑了。

Http1.1以后，Keep-Alive已经默认支持并开启。客户端（包括但不限于浏览器）发送请求时会在Header中增加一个请求头Connection: Keep-Alive，当服务器收到附带有Connection: Keep-Alive的请求时，也会在响应头中添加Keep-Alive。这样一来，客户端和服务器之间的HTTP连接就会被保持，不会断开（断开方式下面介绍），当客户端发送另外一个请求时，就可以复用已建立的连接。

### Keep-alive的value

### Keep-alive的链接限制和规则

1. 在HTTP/1.0中, keep-alive并不是默认使用的。客户端必须发送一个Connection: Keep-Alive请求首部来激活keep-alive连接。

从HTTP/1.1起，浏览器默认都开启了Keep-Alive，保持连接特性，客户端和服务器都能选择随时关闭连接，则请求头中为connection:close。

客户端探明响应中没有Connection: Keep-Alive响应首部,就可以知道服务器发出响应之后是否会关闭连接了。

1. Connection: Keep-Alive首部必须随所有希望保持持久连接的报文一起发送。如果客户端没有发送Connection: Keep-Alive首部,服务器就会在那条请求之后关闭连接。

1、有Keep-Alive 第一次 响应时间一共为: 511

有Keep-Alive 第二次 响应时间一共为: 几十毫秒

图片包含 游戏机, 截图

描述已自动生成

1. 无Keep-Alive:alive 响应时间固定为几百毫秒

3、只有在无需检测到连接的关闭即可确定报文实体主体部分长度的情况下,才能将连接保持在打开状态--也就是说实体的主体部分必须有正确的Content-Length,有多部件媒体类型,或者用分块传输编码的方式进行了编码。在一条keep-alive信道中回送错误的content-Length是很糟糕的事,这样的话,事务处理的另一端就无法精确地检测出一条报文的结束和另一条报文的开始了。

### 链接复用后的包边界问题

由于Http中Header的存在，通过定义一些报文长度的首部字段，可以很方便的处理包边界问题。

在Http中，有两种方式处理包边界问题：

1、Content-Length处理包边界(数据量小)

特点：数据量小

这个是最通常的处理方式，接收端处理报文时首先读取完整首部（Header），然后通过Header中的Content-Length来确认报文大小，读取报文时按此长度读取即可，超出长度的报文（“粘包”）不读取，不够长度的报文缓存等待继续读取（“拆包”）。

2、Chunked处理包边界

当返回的数据比较大时，如果等待生成完数据再传输，这样效率比较低下。相比而言，服务器更希望边生成数据边传输。可以在响应头加上以下字段标识分块传输：

特点: 返回的数据量比较大 ，如果等待生成完数据再传输，这样效率比较低下

Transfer-Encoding: chunked

（1）当选择分块传输时，响应头中可以不包含Content-Length。

（2）服务器会先回复一个不带数据的报文（只有响应行和响应头和\r\n），然后开始传输若干个数据块。每个数据块格式如下：

图片包含 游戏机, 截图

描述已自动生成

（3）当传输完若干个数据块后，需要再传输一个空的数据块

图片包含 游戏机

描述已自动生成

（4） 当客户端收到空的数据块时，则客户端知道数据接收完毕。

### Keep-alive的配置

1、Nginx中设置Keep-Alive（服务端）

Syntax: keepalive\_timeout timeout [header\_timeout];

Default: keepalive\_timeout 75s;

Context: http, server, location

第一个参数设置一个超时，在此期间保持活动的客户机连接将在服务器端保持打开状态。如果为0则禁用保Keep-Alive。

第二个可选参数在“Keep-Alive: timeout=time”响应头字段中设置一个值。

Keep-Alive: timeout=time”报头字段被Mozilla和Konqueror识别。MSIE在大约60秒内自动关闭保持连接。

2、Tomcat中设置Keep-Alive（服务端）

在<Connector>标签中配置属性：

Keep-Alive timeout配置

keepAliveTimeout="超时时间"，默认值是使用为connectionTimeout属性设置的值 。值为-1表示没有（即无限）超时。

Keep-Alive requests（连接可用次数）配置： maxKeepAliveRequests="连接可用次数"，-1为永不失效。如果未指定，默认为100。

例如1:保持20s链接,超过2次链接将失效。重新建立链接。

图片包含 监控, 照片, 屏幕, 钟表

描述已自动生成

3、springboot中设置



图片包含 截图, 鸟

描述已自动生成

4、Apache HttpClient 设置Keep-Alive（客户端）

Apache HttpClient在处理KeepAlive的地方设计的比较灵活，提供了可配置的接口，使用者可以使用Http标准的策略，也自定定制策略。

其他设置(参考chrom文件夹中)

### Keep-alive的哑代理问题

哑代理只是单纯的转发请求，并不能进行解析处理、维持持久连接等其他工作，而聪明的代理可以解析接收到的报文同时可以维持持久连接。

当客户端与服务器之间存在不解析直接转发的代理时，connection：keep-alive这个首部是直接转发给服务器的，服务器接收了这个请求之后，就会向客户端发送带有connection：keep-alive的响应，同样盲代理不会解析响应，直接将全部响应转发回客户端。因为客户端收到了这个首部，就认为建立持久连接已经成功了，但是中间的”笨代理“，并不知道这些事情，

**笨代理只有一种行为模式：在转发请求和回送服务器响应请求之后就认为这次事务结束了，等待连接断开**，而这时由于connection：keep-alive首部已经发送到服务器和客户端，双方都认为持久连接已经建立完成，这样就变成了两边认为持久连接OK而中间的哑代理等待连接断开的情况，这种情况下如果客户端再一次在这条连接上发送请求，请求就会在亚代理处停止，因为哑代理已经在等待连接关闭。这种状态会导致浏览器一直处于挂起状态，直到客户端或服务器之中一个连接超时，关闭连接为止，一段美好的牵手就这么没了（哑代理就是把内容原封不动的转发到代理）。

解决方法

首先客户端发送Proxy-connection首部请求

这是一个非标准请求，也就是说即使服务器接收到这个首部也不知道他是干什么的，在服务器眼中它只知道客户端申请持久连接的首部为connection：keep-alive。

### HTTP/1.1的持久链接

与HTTP/1.0+的keep-alive连接不同, HTTP/1.1持久连接在默认情况下是激活的。除非特别指明,否则HTTP/1.1假定所有连接都是持久的。要在事务处理结束之后将连接关闭, HTTP/1.1应用程序必须向报文中显式地添加一个Connection:close首部。这是与以前的HTTP协议版本很重要的区别,在以前的版本中, keep-alive连接要么是可选的,要么根本就不支持。

HTTP/1.1客户端假定在收到响应后,除非响应中包含了**connection**: close首部,不然HTTP/1.1连接就仍维持在打开状态。但是,客户端和服务器仍然可以随时关闭空闲的连接。不发送Connection : close并不意味着服务器承诺永远将连接保持在打开状态

1、发送了connection: close请求首部之后,客户端就无法在那条连接上发送更多的请求了。

2、如果客户端不想在连接上发送其他请求了,就应该在最后一条请求中发送一个connection: close请求首部。

3、只有当连接上所有的报文都有正确的、自定义报文长度时-也就是说,实体主体部分的长度都和相应的content-Length一致,或者是用分块传输编码方式编码的-连接才能持久保持。

4、HTTP/1.1的代理必须能够分别管理与客户端和服务器的持久连接-每个持久

连接都只适用于一跳传输。

5、(由于较老的代理会转发Connection首部,所以) HTTP/1.1的代理服务器不应该与HTTP/1.0客户端建立持久连接,除非它们了解客户端的处理能力。实际上,这一点是很难做到的,很多厂商都违背了这一原则。

6、尽管服务器不应该试图在传输报文的过程中关闭连接,而且在关闭连接之前至少应该响应一条请求,但不管connection首部取了什么值, HTTP/1.1设备都可以在任意时刻关闭连接。

7、HTTP/1.1应用程序必须能够从异步的关闭中恢复出来。只要不存在可能会累积起来的副作用,客户端都应该重试这条请求。除非重复发起请求会产生副作用,否则如果在客户端收到整条响应之前连接关闭了,客户端就必须要重新发起请求。

8、一个用户客户端对任何服务器或代理最多只能维护两条持久连接,以防服务器过载。代理可能需要更多到服务器的连接来支持并发用户的通信,所以,如果有N个用户试图访问服务器的话,代理最多要维持2N条到任意服务器或父代理的连接。

### 链接的关闭

1、Content-Length及截尾操作

每条HTTP响应都应该有精确的Content-Length首部,用以描述响应主体的尺寸。一些老的HTTP服务器会省略content-Length首部,或者包含错误的长度指示,这样就要依赖服务器发出的连接关闭来说明数据的真实末尾。

客户端或代理收到一条随连接关闭而结束的HTTP响应,且实际传输的实体长度与Content-Length并不匹配(或没有content-Lengeh)时,接收端就应该质疑长度的正确性。

# http的结构

# 代理

## 代理描述

### 为什么使用代理

代理服务器可以实现各种时髦且有用的功能。它们可以改善安全性,提高性能,节省费用。代理服务器可以看到并接触到所有流过的HTTP流量,所以代理可以监视流量并对其进行修改,以实现很多有用的增值Web服务。这里给出了几种代理使用

1、过滤器

小学在为教育站点提供无阻碍访问的同时,可以利用过滤器代理来阻止学生访问成人内容。如图6-3所示,代理应该允许学生无限制地访问教育性内容,但对不适合儿童的站点要强行禁止访问。

2、文档访问控制

可以用代理服务器在大量Web服务器和Web资源之间实现绕一的访问控制第略,创建审核跟踪机制。这在大型企业环境或其他分布式机构中是很有用的。

集中式控制访问代理

客户端1无限制地访问服务器A的新闻页面

客户端2可以无限制地访问因特网.

在允许客户端3访问服务器B之要求其输如口令

3、安全防火墙

网络安全工程师通常会使用代理服务器来提高安全性。代理服务器会在网络中的单一安全节点上限制哪些应用层协议的数据可以流入或流出一个组织。还可以供用来消除病毒的Web和E-mail代理使用的那种挂钩程序,以便对流量进行详细的检查。

Web缓存

代理缓存维护了常用文档的本地副本,并将它们按需提供,以减少缓慢且昂贵的

因特网通信。

反向代理

代理可以假扮Web服务器。这些被称为替代物或反向代理(re-verse proxy)的代理接收发给web服务器的真实请求,但与web服务器不同的是,它们可以发起与其他服务器的通信,以便按需定位所请求的内容。

可以用这些反向代理来提高访问慢速wcb服务器上公共内容时的性能。在这种配置中,通常将这些反向代理称为服务器加速器(server accelerator) (参见图6-7)。还可以将替代物与内容路由功能配合使用,以创建按需复制内容的分布式网络。

转码器

代理服务器在将内容发送给客户端之前,可以修改内容的主体格式。在这些数据表示法之间进行的透明转换被称为转码(transcoding)。

转码代理可以在传输GIF图片时,将其转换成JPEG图片,以减小尺寸。也可以对图片进行压缩,或降低颜色的色彩饱和度以便在电视上观看。同样,可以对文本文件进行压缩,并为能够使用因特网的呼机和智能手机生成小型的文本摘要Web页面。代理甚至可以在传输文档的过程中将其转换成外语。

匿名者

匿名者代理会主动从HTTP报文中删除身份特性(比如客户端IP地址、Erom

首部、 Referer首部、cookie, URI的会话1D),从而提供高度的私密性和匿名性。

匿名代理会对用户报文进行下列修改以增加私密性。

从User-Agent首部删除用户的计算机与os类型。

删除From首部以保护用户的E-mail地址。

删除Referer首部来掩盖用户访问过的其他站点。

删除Cookie首部以剔除概要信息和身份的数据。

### 代理网关的区别

代理使用同一种协议,网关则将不同的协议连接起来。

严格来说,代理连接的是两个或多个使用相同协议的应用程序,而网关连接的则是两个或多个使用不同协议的端点。网关扮演的是“协议转换器”的角色,即使客户端和服务器使用的是不同的协议,客户端也可以通过它完成与服务器之间的事务处理。

图片包含 游戏机

描述已自动生成

实际上,代理和网关之间的区别很模糊。由于浏览器和服务器实现的是不同版本的HTTP,代理也经常要做一些协议转换工作。而商业化的代理服务器也会实现网关的功能来支持SSL安全协议、SOCKS防火墙、FTP访问,以及基于web的应用程序。

## 代理的结构

### 层次结构概念

可以通过代理层次结构（proxy hierarchy）将代理级联起来。如下图所示，在代理的层次结构中，会将报文从一个代理传给另一个代理，直到最终抵达原始服务器为止（然后通代理传回给客户端）

图片包含 游戏机, 衬衫

描述已自动生成

### 代理名词

**父代理：**下一个入口（inbound）代理（靠近服务器）被称为父代理

**子代理：**下一个出口（outbound）代理 （靠近客户端）被称为子代理

图片包含 游戏机

描述已自动生成

## 代理分类

代理服务器可以是某个客户端专用的,也可以是很多客户端共享的。单个客户端专用的代理被称为私有代理。众多客户端共享的代理被称为公共代理。

### 公共代理

大多数代理都是公共的共享代理。集中式代理的费效比更高,更容易管理。某些代理应用,比如高速缓存代理服务器,会利用用户间共同的请求,这样的话,汇人同一个代理服务器的用户越多,它就越有用。

### 私有代理

专用的私有代理并不常见,但它们确实存在,尤其是直接运行在客户端计算机上的时候。有些浏览器辅助产品,以及一些ISP服务,会在用户的PC上直接运行一些小型的代理,以便扩展浏览器特性,提高性能,或为免费ISP服务提供主机广告。

## 流量如何到达代理

### 修改客户端

很多Web客户端,包括网景和微软的浏览器,都支持手工和自动的代理配置。如果将客户端配置为使用代理服务器,客户端就会将HTTP请求有意地直接发送给代理,而不是原始服务器。

客户端代理的设置：

1、显式地设置要使用的代理

2、预先配置浏览器

浏览器厂商或发行商会在将浏览器发送给其客户之前预先对浏览器(或所有其他 web客户端)的代理设置进行手工配置。

3、代理的自动配置( Proxy Auto-Configuration,PAC)

提供一个URI,指向一个用 JavaScript语言编写的代理自动配置文件;客户端会取回这个 JavaScript文件,并运行它以决定是否应该使用一个代理,如果是的话,应该使用哪个代理服务器。

3、WPAD的代理发现

有些浏览器支持Web代理自动发现协议( Web Proxy Autodiscovery Protocol,wPAD),这个协议会自动检测出浏览器可以从哪个“配置服务器”下载到一个自动配置文件。

### 修改网络

网络基础设施可以通过若干种技术手段,在客户端不知道,或没有参与的情况下,拦截网络流量并将其导入代理。这种拦截通常都依赖于监视HTTP流量的交换设备及路由设备,在客户端毫不知情的情况下,对其进行拦截,并将流量导入一个代理(参见图6-14b),这种代理被称为栏截(intercepting)代理。

### 修改DNS的命名空间

放在Web服务器之前的代理服务器—替代物,会直接假扮Web服务器的名字和IP地址,这样,所有的请求就会发送给这些替代物,而不是服务器了(参见图6-14c)。要实现这一点,可以手工编辑DNS名称列表,或者用特殊的动态DNS服务器根据需要来确定适当的代理或服务器。有时在安装过程中,真实服务器的IP地址和名称被修改了,替代物得到的会是之前的地址和名称。

### 修改 Web 服务器

也可以将某些 Web 服务器配置为向客户端发送一条 HTTP 重定向命令（响应码 305），将客户端请求重定向到一个代理上去。收到重定向命令后，客户端会与代 理进行通信（如下图中的d

图片包含 游戏机, 文字, 地图

描述已自动生成

## 代理URL和服务器URL

### 区别

除了一点之外, Web服务器报文和Web代理报文的语法是一样的。客户端向服务器而不是代理发送请求时, HTTP请求报文中的URI会有所不同。

# 缓存

图片包含 游戏机, 文字, 地图

描述已自动生成

图片包含 标志, 游戏机, 钟表, 笔记本

描述已自动生成

## 浏览器缓存

### 浏览器缓存作用

1. 缓存可以减少冗余的数据传输。节省了网络带宽，从而更快的加载页面。

2. 缓存降低了服务器的要求，从而服务器更快的响应。

### memory cache 和 disk cache

memory cache: 它是将资源文件缓存到内存中。等下次请求访问的时候不需要重新下载资源，而是直接从内存中读取数据。

disk cache: 它是将资源文件缓存到硬盘中。等下次请求的时候它是直接从硬盘中读取。

两则的区别:

memory cache(内存缓存)退出进程时数据会被清除，而disk cache(硬盘缓存)退出进程时数据不会被清除。内存读取比硬盘中读取的速度更快。但是我们也不能把所有数据放在内存中缓存的，因为内存也是有限的。

memory cache(内存缓存)一般会将脚本、字体、图片会存储到内存缓存中。

disk cache(硬盘缓存) 一般非脚本会存放在硬盘中，比如css这些。

缓存读取的原理：先从内存中查找对应的缓存，如果内存中能找到就读取对应的缓存，否则的话就从硬盘中查找对应的缓存，如果有就读取，否则的话，就重新网络请求。

### 浏览器更新动作对缓存的影响

所谓用户行为对浏览器缓存的影响，指的就是用户在浏览器如何操作时，会触发怎样的缓存策略。主要有 3 种：

1、打开网页，地址栏输入地址(或者重新打开一个tab)： 查找 disk cache 中是否有匹配。如有则使用；如没有则发送网络请请求。

重新打开一个tab,原有的memory cache 已经释放,不可用。

2、普通刷新 (F5)：因为 TAB 并没有关闭，因此 memory cache 是可用的，会被优先使用(如果匹配的话)。其次才是 disk cache。

1、F5刷新当前页面,但是不会刷新当前页面的其他请求(使用协商缓存)。

2、其他请求先从缓存中获取，获取不到才会访问服务器(使用前缓存)。

3强制刷新 (Ctrl + F5)：浏览器不使用缓存，因此发送的请求头部均带有 Cache-control: no-cache(为了兼容，还带了 Pragma: no-cache),服务器直接返回 200 和最新内容。把浏览器中的临时文件夹的文件删除再重新从服务器下载。

图片包含 游戏机

描述已自动生成

## 缓存位置

从缓存位置上来说分为四种，并且各自有优先级，当依次查找缓存且都没有命中的时候，才会去请求网络。

Service Worker

Memory Cache

Disk Cache

Push Cache

### Service Worker

Service Worker 是运行在浏览器背后的独立线程，一般可以用来实现缓存功能。使用 Service Worker的话，传输协议必须为 HTTPS。因为 Service Worker 中涉及到请求拦截，所以必须使用 HTTPS 协议来保障安全。Service Worker 的缓存与浏览器其他内建的缓存机制不同，它可以让我们自由控制缓存哪些文件、如何匹配缓存、如何读取缓存，并且缓存是持续性的。

Service Worker 实现缓存功能一般分为三个步骤：首先需要先注册 Service Worker，然后监听到 install 事件以后就可以缓存需要的文件，那么在下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件，否则就去请求数据。

当 Service Worker 没有命中缓存的时候，我们需要去调用 fetch 函数获取数据。也就是说，如果我们没有在 Service Worker 命中缓存的话，会根据缓存查找优先级去查找数据。但是不管我们是从 Memory Cache 中还是从网络请求中获取的数据，浏览器都会显示我们是从 Service Worker 中获取的内容。

### Memory Cache

Memory Cache 也就是内存中的缓存，主要包含的是当前中页面中已经抓取到的资源,例如页面上已经下载的样式、脚本、图片等。读取内存中的数据肯定比磁盘快,内存缓存虽然读取高效，可是缓存持续性很短，会随着进程的释放而释放。 一旦我们关闭 Tab 页面，内存中的缓存也就被释放了。

那么既然内存缓存这么高效，我们是不是能让数据都存放在内存中呢？

这是不可能的。计算机中的内存一定比硬盘容量小得多，操作系统需要精打细算内存的使用，所以能让我们使用的内存必然不多。

当我们访问过页面以后，再次刷新页面，可以发现很多数据都来自于内存缓存

图片包含 游戏机, 截图

描述已自动生成

内存缓存中有一块重要的缓存资源是preloader相关指令（例如<link rel="prefetch">）下载的资源。总所周知preloader的相关指令已经是页面优化的常见手段之一，它可以一边解析js/css文件，一边网络请求下一个资源。

需要注意的事情是，内存缓存在缓存资源时并不关心返回资源的HTTP缓存头Cache-Control是什么值，同时资源的匹配也并非仅仅是对URL做匹配，还可能会对Content-Type，CORS等其他特征做校验。

### Disk Cache

Disk Cache 也就是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache 胜在容量和存储时效性上。

在所有浏览器缓存中，Disk Cache 覆盖面基本是最大的。它会根据 HTTP Herder 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。并且即使在跨站点的情况下，相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据。绝大部分的缓存都来自 Disk Cache，关于 HTTP 的协议头中的缓存字段，我们会在下文进行详细介绍。

浏览器会把哪些文件丢进内存中？哪些丢进硬盘中？

关于这点，网上说法不一，不过以下观点比较靠得住：

对于大文件来说，大概率是不存储在内存中的，反之优先

当前系统内存使用率高的话，文件优先存储进硬盘

4.Push Cache

Push Cache（推送缓存）是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被使用。它只在会话（Session）中存在，一旦会话结束就被释放，并且缓存时间也很短暂，在Chrome浏览器中只有5分钟左右，同时它也并非严格执行HTTP头中的缓存指令。

Push Cache 在国内能够查到的资料很少，也是因为 HTTP/2 在国内不够普及。这里推荐阅读Jake Archibald的 HTTP/2 push is tougher than I thought 这篇文章，文章中的几个结论：

所有的资源都能被推送，并且能够被缓存,但是 Edge 和 Safari 浏览器支持相对比较差

可以推送 no-cache 和 no-store 的资源

一旦连接被关闭，Push Cache 就被释放

多个页面可以使用同一个HTTP/2的连接，也就可以使用同一个Push Cache。这主要还是依赖浏览器的实现而定，出于对性能的考虑，有的浏览器会对相同域名但不同的tab标签使用同一个HTTP连接。

Push Cache 中的缓存只能被使用一次

浏览器可以拒绝接受已经存在的资源推送

你可以给其他域名推送资源

如果以上四种缓存都没有命中的话，那么只能发起请求来获取资源了。

那么为了性能上的考虑，大部分的接口都应该选择好缓存策略，通常浏览器缓存策略分为两种：强缓存和协商缓存，并且缓存策略都是通过设置 HTTP Header 来实现的。

## 缓存过期策略

设置响应头（注意浏览器有自己的缓存替换策略，即便资源过期，不一定被浏览器删除。同样资源未过期，可能由于缓存空间不足而被其他网页新的缓存资源所替换而被删除。

### Last-Modified

如果以上均未设置，却设置了 Last-Modified ，浏览器隐式的设置资源过期时间为 (Date - Last-Modified) \* 10% 缓存过期时间。

例:

图片包含 游戏机

描述已自动生成

过期时间 = (当前时间-最后修改时间)\*0.1 = 1分钟。

这个缓存会存留1分钟,1分钟之后会重新请求。当前时间相对于最后修改越大时,缓存的时间越长。

## 校验资源过期方式

HTTP有一些简单的机制可以在不要求服务器记住有哪些缓存拥有其文档副本的情况下,保持已缓存数据与服务器数据之间充分一致。HTTP将这些简单的机制称为文档过期(document expiration)和服务器再验证(server revalidation)。

图片包含 游戏机

描述已自动生成

设置请求头：

1、If-None-Match 如果缓存资源过期，浏览器发起请求会自动把原来缓存响应头里的 ETag 值设置为请求头 If-None-Match 的值发送给服务器用于比较。一般设置为文件的 hash 码或其他标识能够精确判断文件是否被更新，为强校验。

2、If-Modified-Since 同样对应缓存响应头里的 Last-Modified 的值。此值可能取得 ctime 的值，该值可能被修改但文件内容未变，导致对比不准确，为弱校验。

### 强制校验缓存

有时我们既想享受缓存带来的性能优势，可有时又不确认资源内容的更新频度或是其他资源的入口，我们想此服务器资源一旦更新能立马更新浏览器的缓存，这时我们可以设置

Cache-Control: no-cache

### 优先级的问题

如果expires和cache-control同时存在，cache-control会覆盖expires。建议两个都写，cache-control是http1.1的头字段，expires是http1.0的头字段，都写兼容会好点。

例如:此时的expires无效的,浏览器不会进行缓存。

cache-control: no-cache

Expires:时间

## 强缓存

缓存相关的HTTP header 的字段有两个 Expires以及Cache-Control

### 设置 Cache-Control

Cache-Control：no-cache，强制每次请求直接发送给源服务器，而不经过本地缓存版本的校验。这对于需要确认认证应用很有用（可以和public结合使用），或者严格要求使用最新数据 的应用（不惜牺牲使用缓存的所有好处）。

通俗解释：浏览器通知服务器，本地没有缓存数据

**可缓存性**

public

表明响应可以被任何对象（包括：发送请求的客户端，代理服务器，等等）缓存。表示相应会被缓存，并且在多用户间共享。默认是public。

private

表明响应只能被单个用户缓存，不能作为共享缓存（即代理服务器不能缓存它）,可以缓存响应内容。响应只作为私有的缓存，不能在用户间共享。如果要求HTTP认证，响应会自动设置为private。

no-cache

在释放缓存副本之前，强制高速缓存将请求提交给原始服务器进行验证。指定不缓存响应，表明资源不进行缓存。但是设置了no-cache之后并不代表浏览器不缓存，而是在缓存前要向服务器确认资源是否被更改。因此有的时候只设置no-cache防止缓存还是不够保险，还可以加上private指令，将过期时间设为过去的时间。

only-if-cached

表明客户端只接受已缓存的响应，并且不要向原始服务器检查是否有更新的拷贝.

**到期**

max-age=<seconds>

设置缓存存储的最大周期，超过这个时间缓存被认为过期(单位秒)。与Expires相反，时间是相对于请求的时间。max-age会覆盖掉Expires。

s-maxage=<seconds>

覆盖max-age 或者 Expires 头，但是仅适用于共享缓存(比如各个代理)，并且私有缓存中它被忽略。也就是说s-maxage只用于共享缓存，比如CDN缓存（s -> share）。与max-age 的区别是： max-age用于普通缓存，而s-maxage用于代理缓存。如果存在s-maxage,则会覆盖max-age 和 Expires.

max-stale[=<seconds>]

表明客户端愿意接收一个已经过期的资源。 可选的设置一个时间(单位秒)，表示响应不能超过的过时时间。

min-fresh=<seconds>

表示客户端希望在指定的时间内获取最新的响应。

stale-while-revalidate=<seconds>

表明客户端愿意接受陈旧的响应，同时在后台异步检查新的响应。秒值指示客户愿意接受陈旧响应的时间长度。

stale-if-error=<seconds>

表示如果新的检查失败，则客户愿意接受陈旧的响应。秒数值表示客户在初始到期后愿意接受陈旧响应的时间。

重新验证和重新加载

must-revalidate

缓存必须在使用之前验证旧资源的状态，并且不可使用过期资源。表示如果页面过期，则去服务器进行获取。

proxy-revalidate

与must-revalidate作用相同，但它仅适用于共享缓存（例如代理），并被私有缓存忽略。

immutable

表示响应正文不会随时间而改变。资源（如果未过期）在服务器上不发生改变，因此客户端不应发送重新验证请求头（例如If-None-Match或If-Modified-Since）来检查更新，即使用户显式地刷新页面。在Firefox中，immutable只能被用在 https:// transactions.

其他

no-store

缓存不应存储有关客户端请求或服务器响应的任何内容。表示绝对禁止缓存!

no-transform

不得对资源进行转换或转变。Content-Encoding, Content-Range, Content-Type等HTTP头不能由代理修改。例如，非透明代理可以对图像格式进行转换，以便节省缓存空间或者减少缓慢链路上的流量。 no-transform指令不允许这样做。

2、设置 Cache-Control

Cache-Control: max-age=1000 缓存失效时间为1000s之后,缓存有效期之间,不会给服务器发送请求,即使服务器的资源已经不存在。

public和private区别在于是否有中间商赚差价（是否允许CDN代理服务器缓存）

图片包含 游戏机, 截图

描述已自动生成

需要注意的一点是该字段值定义为no-cache并不是说，不准使用缓存，而是需要走接下来的优先级相对较低的另一类--协商缓存。真正决定不用缓存内的资源是将该值定义为no-store

### 设置Exipires时间

设置 Expires ,此时间与本地时间(响应头中的 Date )对比，小于本地时间表示过期，由于本地时钟与服务器时钟无法保持一致，导致比较不精确。

Expires:new Date().toUTCString //格林威治时间格式（GMT)的时间,即世界标准时间。

缺点: 他的缺点很明显，时间期限是服务器生成，存在着客户端和服务器的时间误差，固定时间，HTTP 1.0时的规范。

## 协商缓存

### 协商缓存

协商缓存是通过客户端和服务端进行HTTP通信时，所在响应头和请求头中互相表达“暧昧”的，相互通气，互送缓存标识。

图片包含 游戏机, 文字, 钟表

描述已自动生成

协商缓存失效，返回200和请求结果

图片包含 游戏机, 文字

描述已自动生成

### Last-Modified 和 If-Modified-Since

第一次请求某一个资源时，由于一定不会走缓存，所以服务器端会在资源的响应头中加上一个形如Last-Modified:Mon, 26 Feb 2018 06:37:41 GMT的字段告诉客户端浏览器，这个资源上次最后修改的时间；刷新页面再次请求，这时候的协商缓存会在请求头中加上一个形如If-Modified-Since:Mon, 26 Feb 2018 06:37:41 GMT，字面翻译就是，是否在上个“暧昧”时间后修改了，值毫无疑问是服务端上一次响应给他的时间，让服务器去判断是否在此时间之后资源内容发生了变化。

整个过程也很简单，最后的结果也很简短。如果服务端发现改变了资源，就伴着200的 statuscode 和新鲜的资源给到客户端，若是没有修改，304 Not Modified让客户端从缓存中取。

缺点

图片包含 游戏机, 截图

描述已自动生成

### ETag(实体标签)与If-None-Match

Cache-Control: max-age=20 设置缓存文件的过期时间。

ETag:响应头中,服务器将改文件的标识放到响应头中。

If-None-Match:请求头中,浏览器拿着缓存文件的etag标签到服务器比较。

浏览器第一次请求,获取到文件,在缓存时间之内再次请求,使用的是缓存文件。超过缓存时间的第一次请求,浏览器写到标签到服务器,服务器用这个标签和服务器的标签比较,如果标签不一致返回文件,如果标签一致,返回304,告诉浏览器该文件没有变化,仍然可以使用缓存。

304状态码:如果客户端发送了一个带条件的GET 请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变，则服务器应当返回这个304状态码。简单的表达就是：服务端已经执行了GET，但文件未变化。

解决的问题:

(1)Last-Modified标注的最后修改只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，它将不能准确标注文件的修改时间

(2)、如果某些文件被修改了，但是内容并没有任何变化，而Last-Modified却改变了，导致文件没法使用缓存

(3)、有可能存在服务器没有准确获取文件修改时间，或者与代理服务器时间不一致等情形

图片包含 游戏机, 文字, 截图, 地图

描述已自动生成

### 优先级

Etag 和 If-None-Match的优先级要高于Last-Modified 和 If-Modified-Since，进而会衍生出一个思考，二者相比功能相同，但是表达形式决定了 Etag 解决了 Last-Modified 存在的一些问题，比如Last-Modified 是比较时间，精确到秒，若是毫秒级的改变则没法兼顾，存在着周期性更改的资源，然而有可能资源本身的内容并没有改变，那如果重新请求响应意义并不是那么的大。所以不难理解Etag具有高优先级有他的合理之处。

### tomcat设置

### nginx设置

## 缓存的处理步骤

### 接收

缓存检测到一条网络连接上的活动,读取输入数据。高性能的缓存会同时从多条输入连接上读取数据,在整条报文抵达之前开始对事务进行处理。

### 解析

缓存将请求报文解析为片断,将首部的各个部分放入易于操作的数据结构中。这样,缓存软件就更容易处理首部字段并修改它们了。

### 查找

在第三步中,缓存获取了URL,查找本地副本。本地副本可能存储在内存、本地磁盘,甚至附近的另一台计算机中。专业级的缓存会使用快速算法来确定本地缓存中是否有某个对象。如果本地没有这个文档,它可以根据情形和配置,到原始服务器或父代理中去取,或者返回一条错误信息。

已缓存对象中包含了服务器响应主体和原始服务器响应首部,这样就会在缓存命中时返回正确的服务器首部。已缓存对象中还包含了一些元数据(metadata),用来记录对象在缓存中停留了多长时间,以及它被用过多少次等。

### 新鲜度检测

HTTP通过缓存将服务器文档的副本保留一段时间。在这段时间里,都认为文档是“新鲜的”,缓存可以在不联系服务器的情况下,直接提供该文档。但一旦已缓存副本停留的时间太长,超过了文档的新鲜度限值(freshness limit),就认为对象“过时”了,在提供该文档之前,缓仔要再次与服务器进行确认,以查看文档是否发生了变化。客户端发送给缓存的所有请求首部自身都可以强制缓存进行再验证,或者完全避免验证,这使得事情变得更加复杂了。

### 发送

一旦响应首部准备好了,缓存就将响应回送给客户端。和所有代理服务器一样,代理缓存要管理与客户端之间的连接。高性能的缓存会尽力高效地发送数据,通常可以避免在本地缓存和网络1/0缓冲区之间进行文档内容的复制。

### 创建响应

我们希望缓存的响应看起来就像来自原始服务器的一样,缓存将已缓存的服务器响应首部作为响应首部的起点。然后缓存对这些基础首部进行了修改和扩充。

注意,缓存不应该调整Date首部。 Date首部表示的是原始服务器最初产生这个对象的日期。

### 日志

大多数缓存都会保存日志文件以及与缓存的使用有关的一些统计数据。每个缓存事务结束之后,缓存都会更新缓存命中和未命中数目的统计数据(以及其他相关的度量值),并将条目插入一个用来显示请求类型、URL和所发生事件的日志文件。

### 缓存处理流程图

# 客户端识别和cookie机制

## User-Agent

### 什么是User-Agent

User-Agent中文名为用户代理，简称 UA，是Http协议中的一部分，属于头域的组成部分，它是一个特殊字符串头，使得服务器能够识别客户使用的操作系统及版本、CPU 类型、浏览器及版本、浏览器渲染引擎、浏览器语言、浏览器插件等。在网络请求当中，User-Agent 是标明身份的一种标识，通过这个标识，用户所访问的网站可以显示不同的排版从而为用户提供更好的体验或者进行信息统计；例如用手机访问谷歌和电脑访问是不一样的，这些是谷歌根据访问者的UA来判断的。UA可以进行伪装。

### 常见User-Agent介绍

浏览器UA 字串的标准格式为： 浏览器标识 (操作系统标识; 加密等级标识; 浏览器语言) 渲染引擎标识 版本信息

例如:Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.138 Safari/537.36

User-Agent通常格式：浏览器标识 (操作系统标识; 加密等级标识; 浏览器语言) 渲染引擎标识 版本信息

Mozilla/5.0 ：以前用于Netscape浏览器，目前大多数浏览器UA都会带有

Windows NT 10.1：代表windows10系统 64位

AppleWebKit/537.36：浏览器内核

KHTML: 一个HTML排版引擎

like Gecko：这不是Geckeo 浏览器，但是运行起来像Geckeo浏览器

Chrome/36.0.1985.125: Chrome版本号

User-Agent字段解释

第一部分：Mozilla/5.0

由于历史上的浏览器大战，当时想获得图文并茂的网页，就必须宣称自己是 Mozilla 浏览器。此事导致如今User-Agent里通常都带有Mozilla字样，出于对历史的尊重，大家都会默认填写该部分。

### UA价值分析

1.用来识别是否爬虫：

User-Agent值是用来帮助服务器识别用户使用的操作系统、浏览器、浏览器版本等等信息的，因此也常被用来检测爬虫。许多网站会ban掉来自爬虫的请求，来达到反爬的目的。

服务器可以轻易识别出该请求是否来自爬虫。因此为了减小爬虫被ban的几率，我们可以通过设置请求的User-Agent来达到欺骗服务器的目的。

2.用来判断是否虚假设备：

UA是用户的一种标识，带有浏览器设备等信息，但是UA信息可以篡改，如果某一时间某一地域大批量UA聚集或者说UA标识信息和设备信息不同，可以细分查看是否是虚假设备。

## Referer

## Cookie

### Cookie是什么

cookie翻译过来是“饼干，甜品”的意思，cookie在网络应用中到处存在，当我们浏览之前访问过的网站，网页中可能会显示：你好，王三少，这就会让我们感觉很亲切，像吃了一块很甜的饼干一样。

由于http是无状态的协议，一旦客户端和服务器的数据交换完毕，就会断开连接，再次请求，会重新连接，这就说明服务器单从网络连接上是没有办法知道用户身份的。怎么办呢？那就给每次新的用户请求时，给它颁发一个身份证（独一无二）吧，下次访问，必须带上身份证，这样服务器就会知道是谁来访问了，针对不同用户，做出不同的响应。，这就是Cookie的原理。

其实cookie是一个很小的文本文件，是浏览器储存在用户的机器上的。Cookie是纯文本，没有可执行代码。储存一些服务器需要的信息，每次请求站点，会发送相应的cookie，这些cookie可以用来辨别用户身份信息等作用。

图片包含 游戏机, 截图

描述已自动生成

如图所示,用户首次访问服务器，服务器会返回一个独一无二的识别码；id=23451，这样服务器可以用这个码跟踪记录用户的信息，（购物历史，地址信息等）。

cookie可以包含任意的信息，不仅仅是id，客户端会记录服务器返回来的Set-Cookie首部中的cookie内容。并将cookie存储在浏览器的cookie数据库中，当用户访问同一站点时，浏览器就会挑选当时该站点颁发的id=XXX的身份证（cookie），并在Cookie请求首部发送过去。

### Cookie存在形式

1、Cookie请求头

Cookie头由客户端发送，包含在HTTP请求的头部中。注意，只有cookie的domain和path与请求的URL匹配才会发送这个cookie。

格式:Cookie:name=value;name1=value1;

2、set-cookie响应头

Set-Cookie: qwerty=219ffwef9w0f; Domain=somecompany.co.uk; Path=/; Expires=Wed, 30 Aug 2019 00:00:00 GMT

### cookie的类型

可以按照过期时间分为两类：会话cookie和持久cookie。会话cookie是一种临时cookie，用户退出浏览器，会话Cookie就会被删除了，持久cookie则会储存在硬盘里，保留时间更长，关闭浏览器，重启电脑，它依然存在，通常是持久性的cookie会维护某一个用户周期性访问服务器的配置文件或者登录信息。

持久cookie 设置一个特定的过期时间（Expires）或者有效期（Max-Age）

Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2019 07:28:00 GMT;

### cookie的属性

cookie的过期时间(可选)

http1.0

expires选项用来设置“cookie 什么时间内有效”。

格式: expires= GMT 格式的时间

例如: Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2019 07:28:00 GMT;

意为:该cookie将在2016年2月25日4:18分之后失效，对于失效的cookie浏览器会清空。

图片包含 游戏机

描述已自动生成

注:如果没有设置该选项，则默认有效期为session，即会话cookie。这种cookie在浏览器关闭后就没有了。

图片包含 游戏机, 桌子

描述已自动生成

http2.0

在2.0中可以使用max-age代替两者的作用都是限制cookie 的有效时间。expires的值是一个时间点（cookie失效时刻= expires），而max-age 的值是一个以秒为单位时间段（c

cookie失效时刻= 创建时刻+ max-age。

max-age的默认值是-1

max-age=负数,表示该有效期为一次会话。

max-age=0,表示删除cookie,其实也是表示一次会话。

max-age=正数, 有效期为创建时刻+ max-age

如果给cookie设置一个过去的时间，浏览器会立即删除该cookie。

cookie的domain:

产生Cookie的服务器可以向set-Cookie响应首部添加一个Domain属性来控制哪些站点可以看到那个cookie

1、本级域名的cookie可以被下级域名所共享。下级域名请求时会携带上级域名的cookie。

2、同级域名的cookie相互独立,不共享。

3、不同级的域名下的cookie的名字可以相同,同级域名下的cookie相同名称多会被覆盖。

例如:

域名1设置domain为libin.com.cn的cookie,mykey=myvalue1。

域名2设置domain为yun.libin.com.cn的cookie,mykey=myvalue2。

服务器域名2都可拿到相同的mykey。服务端可拿到多个相同名称的cookie,但是没有办法做区分,请求头无法携带cookie的domain等信息。

4、cookie的路径 Path

cookie必须在同一网站下且cookie的path路径为当前url或者是当前url的父级时才是共享的。

Path: 表示这个cookie影响到的路径，浏览器跟会根据这项配置，像指定域中匹配的路径发送cookie。

1、如果不设置cookie的路径,默认path是当前目录的上级目录

例如:

/api/test/cookie : /api/test

/api : /

2、子目录能获取到父级目录的cookie,父级目录的cookie在子目录下共享。

5、cookie的secure

secure:

true:只能在https连接中传输http链接不会传输。

false: HTTP、HTTPS连接都可以传输

1、只有通过https请求时,才会携带cookie？

设置了属性secure，cookie只有在https协议加密情况下才会发送给服务端。但是这并不是最安全的，由于其固有的不安全性，敏感信息也是不应该通过cookie传输的.

如果设置了这个属性，那么只会在 SSH 连接时才会回传该 Cookie

Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure;

HttpOnly

1、只有通过https请求时,才会携带cookie？

2、设置HttpOnly=true的cookie不能被js获取到, 无法用document.cookie打出cookie的内容。

例如:Set-Cookie: zzz=11; Expires=Wed, 03-Jun-2020 08:47:59 GMT; HttpOnly

### 服务端解析Cookie

## 保留户信息

### 请求头

### From

### User-Agent

## 跨域问题

# http认证

## Basic认证

### basic认证简介

Basic认证是一种较为简单的HTTP认证方式，客户端通过明文（Base64编码格式）传输用户名和密码到服务端进行认证，通常需要配合HTTPS来保证信息传输的安全。

Web应用程序收到一条HTTP请求报文时,服务器没有按照请求执行动作,而是以一个“认证质谕”进行响应,要求用户提供一些保密信息来说明他是谁,从而对其进行质询。

用户再次发起请求时,要附上保密证书(用户名和密码),如果证书不匹配,服务器可以再次质询客户端,或产生一条错误信息。如果证书匹配,就可以正常完成请求了。

### Basic认证头信息

1、服务器响应头

WWW-Authenticate: Basic realm="favccxx.com"

### 认证流程

1、 客户端向服务器请求访问受保护的数据，请求的内容可能是一个网页或者是一个其它的MIME类型，此时，假设客户端尚未被验证，则客户端提供如下请求至服务器:

当请求的资源需要BASIC认证时,服务器会随状态码401Authorization Required,返回带www-Authenticate首部字段的响应。该字段内包含认证的方式(BASIC)及Request-URI安全域字符串(realm)

2、服务器向客户端发送验证请求代码401,服务器返回的数据大抵如下：

图片包含 游戏机, 截图

描述已自动生成浏览器识别到Basic后弹出对话框Realm表示Web服务器中受保护文档的安全（比公司财务信息域和公司员工信息域），用来指示需要哪个域的用户名和密码。

3、当符合http1.0或1.1规范的客户端（如IE，FIREFOX）收到401返回值时，将自动弹出一个登录窗口，要求用户输入用户名和密码。

图片包含 游戏机, 截图

描述已自动生成

4、用户输入用户名和密码后，将用户名及密码以BASE64加密方式加密，并将密文放入前一条请求信息中，则客户端发送的第一条请求信息则变成如下内容：

编码方式:将admin:passportword base64编码。浏览器会自动完成编码的转换。

例如:123456:123456 编码后为 MTIzNDU2OjEyMzQ1Ng==

图片包含 截图, 游戏机

描述已自动生成

### Tomcat进行Basic认证

## DIGEST认证(摘要认证)

### 基本概述

基本认证便捷灵活，但极不安全。用户名和密码都是以明文形式（只用Base64编码进行了扰码，没有任何防止用户攻击的手段）传送的。

摘要是“对信息主体的浓缩”。摘要是一种单向函数,主要用于将无限的输入值转换为有限的浓缩输出值。,常见的摘要函数MD5,\一个128位的摘要。

### 优缺点

相对于basic认证的改进

1、永远不会以明文方式在网络上发送密码。

2、可以防止恶意用户捕获并重放认证的握手过程。

3、可以有选择地防止对报文内容的篡改。

4、防范其他几种常见的攻击方式。

首先基本认证的一个重大缺陷就是在请求中使用了Base64加密来传输了用户名和密码，Base64是对称加密，也就意味着如果有人截获了请求，那么就相当于用户名和密码全部暴露。

所以摘要认证的基本原则就是绝不通过网络发送明文密码，而是发送一个密码的摘要信息来取代密码，并且这个摘要信息是不可逆的，也就是我们需要用非对称加密算法来加密。例如md5加密

比如客户端需要加密的信息是hello world，经过md5加密后请求发送的是5eb63bbbe01eeed093cb22bb8f5acdc3这一串密文，任何人截获到这段密文是无法反推回hello world的，然后服务端收到这个密文后。他先去数据库中找到你存储的密文hello world（当然是根据你传过来的未加密的用户名来找的，我们省略了没说），然后服务端也对hello world进行md5加密，对比加密后的字符串和客户端发过来的，当然一样啦，那就验证通过了。

当然，这里会有一个严重的问题，黑客截获了摘要信息和密码一样好用，他只要伪造请求，然后把摘要信息再发给服务端就照样可以为所欲为了。那么怎么办呢？加随机数就可以解决。服务端第一次发一个随机串randomStr给客户端，当然，服务端自己也记下来我发了这个随机串给客户端，然后客户端拿到randomStr后，附在密码后面再执行md5加密，然后再把密文发给服务端。这时候就很好了，服务端也是会先把随机串附在密码后面，然后md5加密得到密文，来和客户端发过来的比较。但是现在即使这个密文被截获了也没用，因为这个密文是根据一个服务端的随机数来生成的，而这个随机数服务端可以设置每次请求或者每一毫秒都不一样，而黑客却无法得知。

### 认证流程

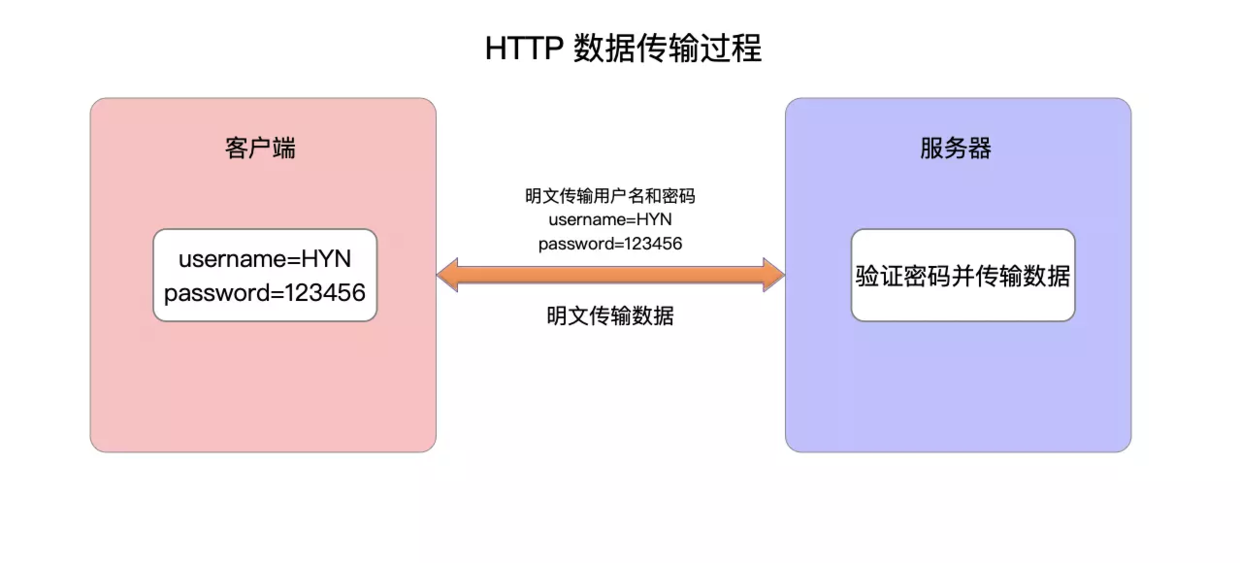
1、请求需认证的资源时,服务器会随着状态码401 AuthorizationRequired,返回带www-Authenticate首部字段的响应。

响应头内容

# https

## 引入

### http



http的缺点:

1、通信使用明文（不加密），内容可能被窃听

由于HTTP本身不具备加密的功能，所以也无法做到对通信整体（使用HTTP协议通信的请求和响应的内容）进行加密。即，HTTP报文使用明文（指未经过加密的报文）方式发送。

对称加密:加密的密钥可能被获取到,密码任然能够拿到。

非对称加密:

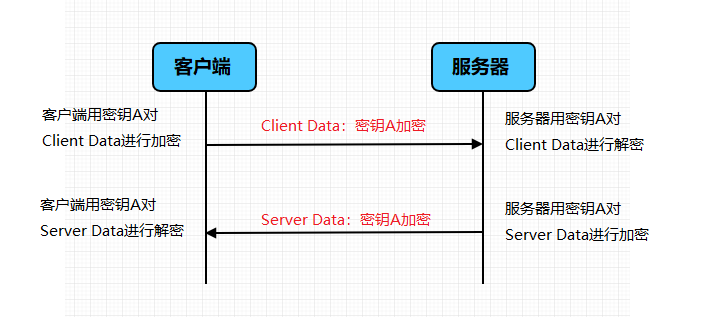
2、HTTP明文协议的缺陷是导致数据泄露、数据篡改、流量劫持、钓鱼攻击等安全问题的重要原因。HTTP协议无法加密数据，所有通信数据都在网络中明文“裸奔”。通过网络的嗅探设备及一些技术手段，就可还原HTTP报文内容。

3、无法证明报文的完整性，所以可能遭篡改:运营商劫

4、没有用户认证。

通信方的身份不会被验证，因此HTTP信息交换的双方都可能被伪装，即没有用户认证。

### 对称加密



对称加密算法的加密和解密都是用**同一个密钥**。

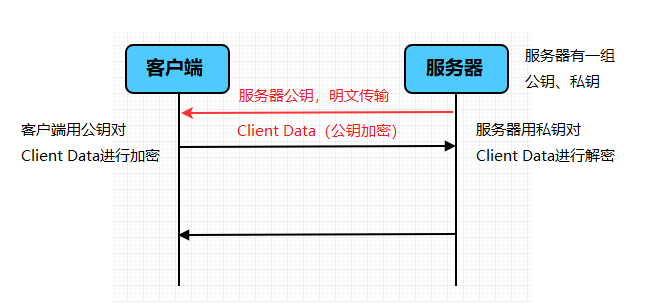
如果通信双方都各自持有同一个密钥，且没有别人知道，则两方的通信安全是可以被保证的（除非密钥被破解）。

然而，最大的问题就是这个密钥怎么让传输的双方知晓，同时不被别人知道。如果由服务器生成一个密钥并传输给浏览器，这个传输过程中密钥被别人劫持，之后他就能用密钥解开双方传输的任何内容。

如果浏览器内部预存了网站A的密钥，且可以确保除了浏览器和网站A，不会有任何外人知道该密钥，那理论上用对称加密是可以的。这样，浏览器只要预存好世界上所有HTTPS网站的密钥就可以了。显然，这样做是不现实的。

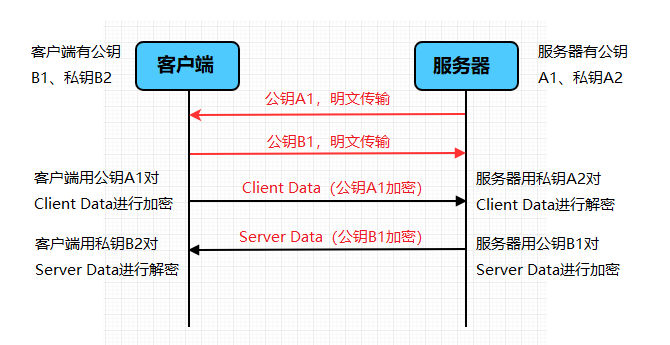
怎么办？解决这个问题，我们就需要非对称加密。

### 非对称加密



基于对称加密存在的问题，又有了非对称加密。非对称加密算法需要一组密钥对，分别是公钥和私钥，这两个密钥是成对出现的。公钥加密的内容需要对应的私钥解密，私钥加密的内容需要对应的公钥解密。私钥由服务器自己保存，公钥发送给客户端。客户端拿到公钥后可以对请求进行加密后发送给服务端，这时候就算中间被截获，没有私钥也无法解密发送的内容，这样确保了客户端发送到服务端数据的安全。

### 非对称加密改良方案



通过一组公钥私钥，已经可以保证单个方向传输的安全性，那用两组公钥私钥，是不是就能保证双向传输都安全了？请看下面的过程：

1、某网站拥有用于非对称加密的公钥A1、私钥A2；浏览器拥有用于非对称加密的公钥B1、私钥B2。

2、浏览器向网站服务器请求，服务器把公钥A1明文传输给浏览器。

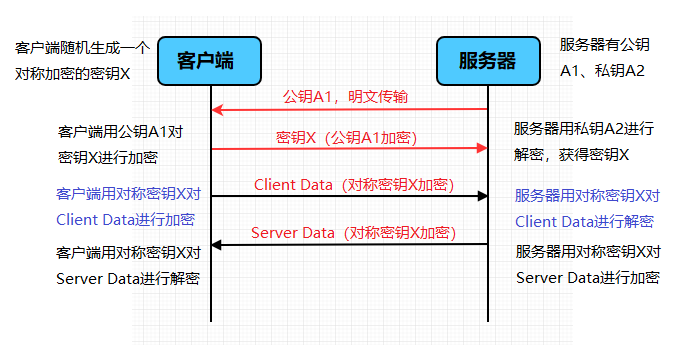
3、浏览器把公钥B1明文传输给服务器。

4、之后浏览器向服务器传输的所有东西都用公钥A1加密，服务器收到后用私钥A2解密。由于只有服务器拥有私钥A2进行解密，所以能保证这条数据的安全。

5、服务器向浏览器传输的所有东西都用公钥B1加密，浏览器收到后用私钥B2解密。同上也可以保证这条数据的安全。

可见确实可行。抛开这里面仍有的漏洞不谈（下文会讲），HTTPS的加密却没使用这种方案，为什么？最主要的原因是非对称加密算法非常耗时，特别是加密解密一些较大数据的时候有些力不从心。而对称加密快很多。那我们能不能运用非对称加密的特性解决前面提到的对称加密的问题？

### 非对称加密 + 对称加密



既然非对称加密耗时，我们考虑是否可以采用非对称加密+对称加密结合的方式，而且要尽量减少非对称加密的次数。

非对称加密、解密各只需一次的方法：

1、某网站拥有用于非对称加密的公钥A1、私钥A2。

2、浏览器向网站服务器请求，服务器把公钥A1明文给传输浏览器。

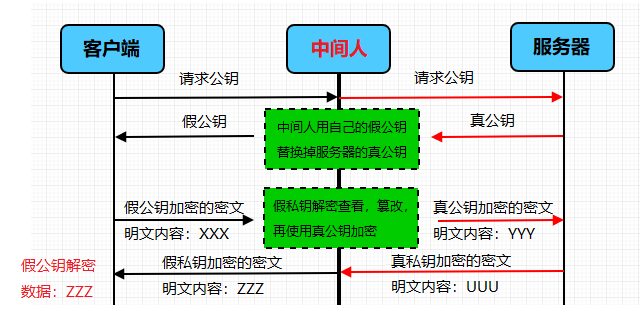
3、浏览器随机生成一个用于对称加密的密钥X，用公钥A1加密后传给服务器。

4、服务器拿到后用私钥A2解密得到密钥X。

5、这样双方就都拥有密钥X了，且别人无法知道它。之后双方所有数据都用密钥X加密解密即可。

HTTPS基本就是采用了这种方案。但还是有漏洞的。

### 中间人攻击



中间人的确无法得到浏览器生成的对称密钥X，这个密钥本身被公钥A1加密，只有服务器才能用私钥A2进行解密。然而中间人却完全不需要拿到私钥A2就能劫持信息，请看：

1、某网站拥有用于非对称加密的公钥A1、私钥A2。

2、浏览器向网站服务器请求，服务器把公钥A1明文传输给浏览器。

3、中间人劫持到公钥A1，保存下来，把数据包中的公钥A1替换成自己伪造的公钥B1（它当然也拥有公钥B1对应的私钥B2）。

4、浏览器随机生成一个用于对称加密的密钥X，用公钥B1（浏览器不知道公钥被替换了）加密后传给服务器。

5、中间人劫持后用私钥B2解密得到密钥X，再用公钥A1加密后传给服务器。

6、服务器拿到后用私钥A2解密得到密钥X。

这样在双方都不会发现异常的情况下，中间人得到了对称密钥X。根本原因是浏览器无法确认自己收到的公钥是不是网站自己的。那么下一步就是解决这个问题：如何证明浏览器收到的公钥一定是该网站的公钥？

## https

### 什么是https

你还记得 HTTP 是怎么定义的吗？HTTP 是一种 超文本传输协议(Hypertext Transfer Protocol) 协议，它 是一个在计算机世界里专门在两点之间传输文字、图片、音频、视频等超文本数据的约定和规范，那么我们看一下 HTTPS 是如何定义的

HTTPS 的全称是 Hypertext Transfer Protocol Secure，它用来在计算机网络上的两个端系统之间进行安全的交换信息(secure communication)，它相当于在 HTTP 的基础上加了一个 Secure 安全的词眼，那么我们可以给出一个 HTTPS 的定义：HTTPS 是一个在计算机世界里专门在两点之间安全的传输文字、图片、音频、视频等超文本数据的约定和规范。 HTTPS 是 HTTP 协议的一种扩展，它本身并不保传输的证安全性，那么谁来保证安全性呢？在 HTTPS 中，使用传输层安全性(TLS)或安全套接字层(SSL)对通信协议进行加密。也就是 HTTP + SSL(TLS) = HTTPS。、



### HTTPS 的作用

加密(Encryption)， HTTPS 通过对数据加密来使其免受窃听者对数据的监听，这就意味着当用户在浏览网站时，没有人能够监听他和网站之间的信息交换，或者跟踪用户的活动，访问记录等，从而窃取用户信息。

数据一致性(Data integrity)，数据在传输的过程中不会被窃听者所修改，用户发送的数据会完整的传输到服务端，保证用户发的是什么，服务器接收的就是什么。

身份认证(Authentication)，是指确认对方的真实身份，也就是证明你是你（可以比作人脸识别），它可以防止中间人攻击并建立用户信任。

### 网站是否使用https

有了上面三个关键指标的保证，用户就可以和服务器进行安全的交换信息了。那么，既然你说了 HTTPS 的种种好处，那么我怎么知道网站是用 HTTPS 的还是 HTTP 的呢？给你两幅图应该就可以解释了。



HTTPS 协议其实非常简单，RFC 文档很小，只有短短的 7 页，里面规定了新的协议名，默认端口号443，至于其他的应答模式、报文结构、请求方法、URI、头字段、连接管理等等都完全沿用 HTTP，没有任何新的东西。

也就是说，除了协议名称和默认端口号外（HTTP 默认端口 80），HTTPS 协议在语法、语义上和 HTTP 一样，HTTP 有的，HTTPS 也照单全收。那么，HTTPS 如何做到 HTTP 所不能做到的安全性呢？关键在于这个 S 也就是 SSL/TLS 。

### 数字证书



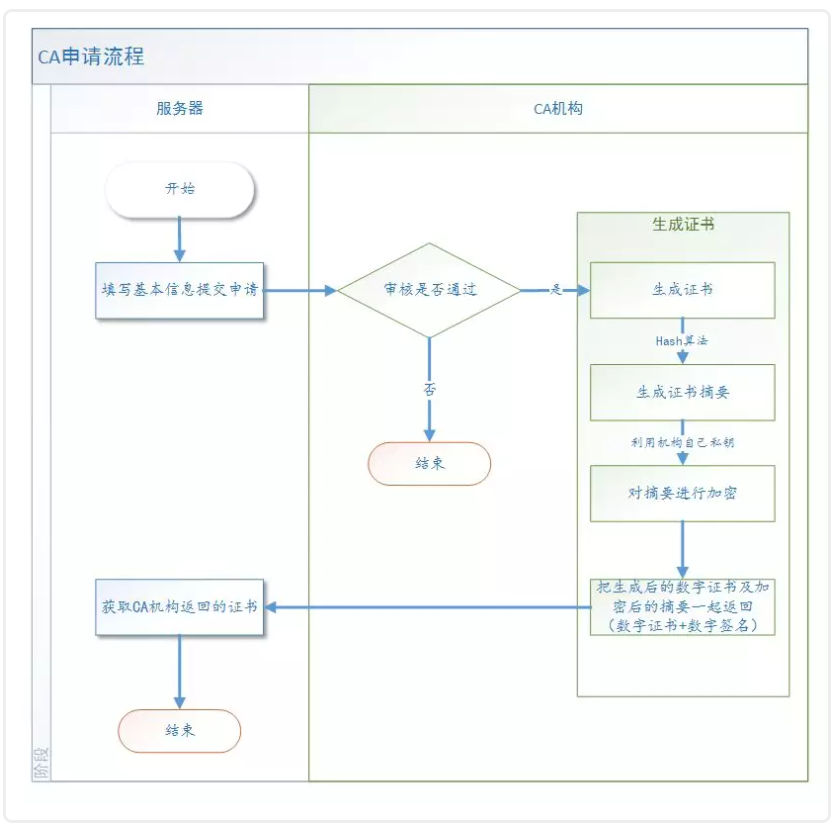
客户端第一次与服务器进行通信的时候，服务器需要出示自己的数字证书，证明自己的身份以及自己的公钥，类似如下（实际上就是一堆数据，这里为了直观）

CA机构就是数字证书颁发的权威机构，负责颁发证书以及验证证书的合法性。

服务器向CA机构提交申请，需要提交站点的信息如域名、公司名称、公钥等等，CA审批无误之后就可以给服务器颁发证书了！

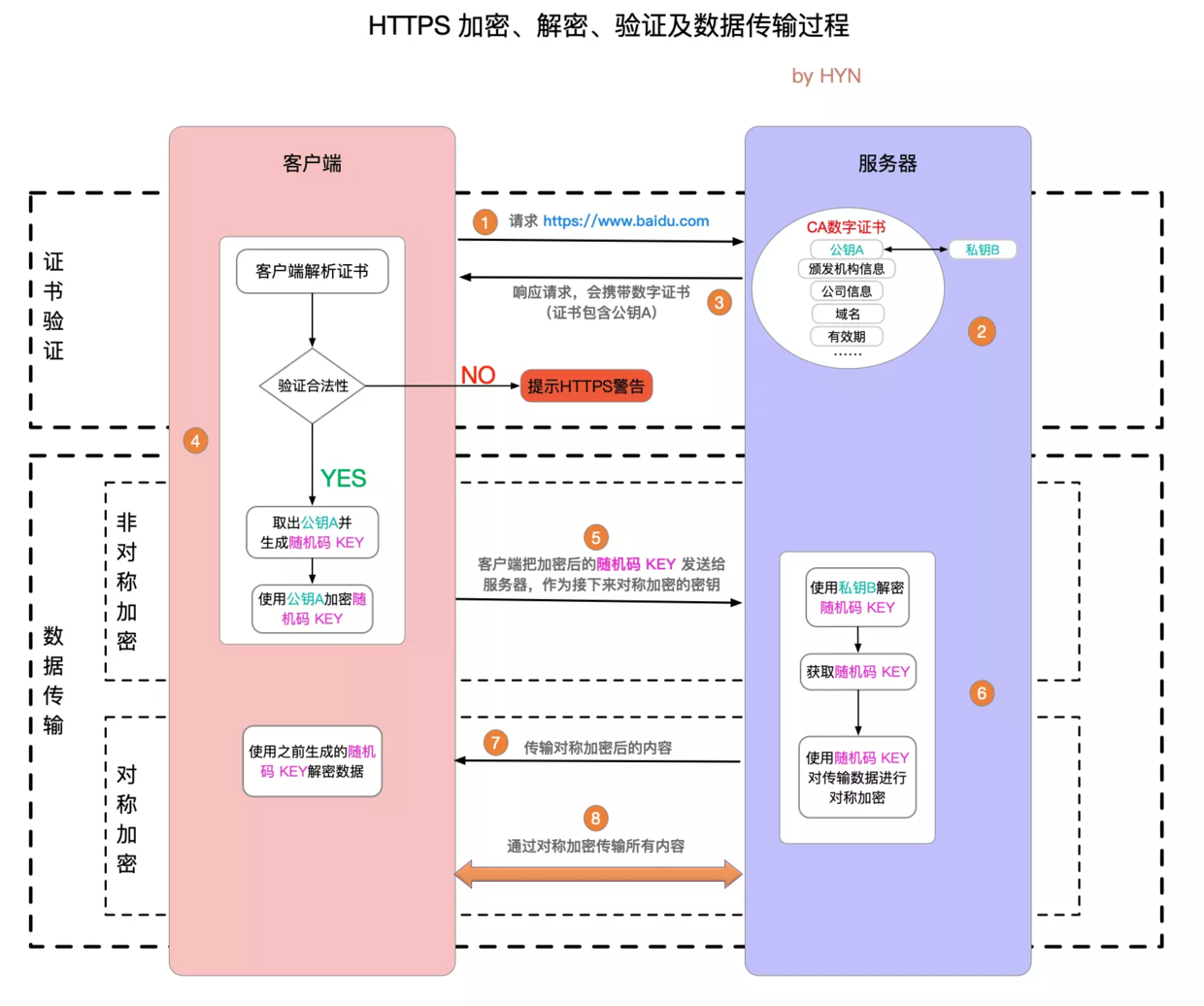
客户端在拿到服务器的证书后，就需要验证证书编号是否能在对应的CA机构查到，并且核对证书的基本信息如证书上的域名是否与当前访问的域名一致等等，还可以拿到证书中服务器的公钥信息用于协商对称密钥！

### 数字签名



服务器提交自己的基本信息想CA机构提出申请，CA机构在给服务器颁发证书的时候，会连同数字证书以及根据证书计算的摘要一同发送给服务器，且这个摘要是需要经过CA机构自己的私钥进行加密的

### https的认证流程



1、服务端将自己域名、邓白氏编码、营业执照 注册到证书机构，证书机构会要求服务段修改dns或下发文件到服务器根路径，确认该服务器是申请者的。证书机构将信息写入到证书中，并通过证书机构的私钥对证书进行加密。向服务端下发数字证书和数字签名。第三方无法去伪造证书,无法通过证书机构的证明。

2、客户端请求服务端，服务端直接下发证书(数字证书+数字签名)，客户端从系统中获取内置的CA机构公钥，对加密的摘要进行解密。然后通过hash 算法对证书内容进行运算。对比两个hash 值是相等，如果相同，再校验域名,有效期信息。然后认证成功。

3、客户端生成随机码key,使用公钥对key进行加密，发送给服务端，作为接下来对称加密的密钥。

4、客户端与服务端通过对称加密所有内容,进行通讯。

1、中间第三方无法伪造证书,内置的系统不存在该证书的公钥，解不开伪造的证书。

2、中间第三方无法伪造服务器,第三方无法解开客户端的证书，不存在证书的私钥。

3、使用对称加密的原因是非对称加密的效率高。

4、如果是https ，客户端会先访问443端口，服务端https的端口为443,如果访问的是非http，服务端需要做一个重定向，重定向到443接口。这里的重定向地址可能被修改为钓鱼网站，下发自己的证书。使用307 重定向，浏览器只会重定向到当前域名的主域名下。 传输数据的端口仍然是80端口。’

### https的四次握手

## SSL/Tsl

### 什么是 SSL/TLS

TLS(Transport Layer Security) 是 SSL(Secure Socket Layer) 的后续版本，它们是用于在互联网两台计算机之间用于身份验证和加密的一种协议。

我们都知道一些在线业务（比如在线支付）最重要的一个步骤是创建一个值得信赖的交易环境，能够让客户安心的进行交易，SSL/TLS 就保证了这一点，SSL/TLS 通过将称为 X.509 证书的数字文档将网站和公司的实体信息绑定到加密密钥来进行工作。每一个密钥对(key pairs) 都有一个 私有密钥(private key) 和 公有密钥(public key)，私有密钥是独有的，一般位于服务器上，用于解密由公共密钥加密过的信息；公有密钥是公有的，与服务器进行交互的每个人都可以持有公有密钥，用公钥加密的信息只能由私有密钥来解密。

### 什么是X.509

X.509 是公开密钥证书的标准格式，这个文档将加密密钥与（个人或组织）进行安全的关联。

X.509 主要应用如下

SSL/TLS 和 HTTPS 用于经过身份验证和加密的 Web 浏览

通过 S/MIME 协议签名和加密的电子邮件

代码签名：它指的是使用数字证书对软件应用程序进行签名以安全分发和安装的过程。

https的认证流程

# 实体

## 报文实体

### 概述

实体首部指出这是一个纯文本文档(content-Type:text/plain),它只有18个字节长(content-Length: 18),和往常一样,一个空白行(CRLF)把首部字段同主体的开始部分分隔开来。

实体首部字段是包含在请求报文和响应报文中的实体部分所使用的首部,用于补充内容的更新时间等与实体相关的信息。

1 content-Length:实体的大小

### 实体首部

Content-Type实体中所承载对象的类型。

Content-Length所传送实体主体的长度或大小。

Content-Language与所传送对象最相配的人类语言。

Content-Encoding对象数据所做的任意变换(比如,压缩)。

Content-Location一个备用位置,请求时可通过它获得对象。

Content-Range如果这是部分实体,这个首部说明它是整体的哪个部分。

Content-MD5实体主体内容的校验和。

Last-Modified所传输内容在服务器上创建或最后修改的日期时间。

Expires实体数据将要失效的日期时间。

Allow该资源所允许的各种请求方法,例如, GET和HEAD

ETag这份文档特定实例(参见15.7节)的唯一验证码。 ETag首部没有正式定义为实体首部,但它对许多涉及实体的操作来说,都是一个重要的首部。

Cache-Control指出应该如何缓存该文档。和ETag首部类似, cache-Control首部也没有正式定义为实体首部。

## Allow

## Content-Type(请求|响应)

### Content-Type简述

Content-Type（MediaType），即是Internet Media Type，互联网媒体类型，也叫做MIME类型。在互联网中有成百上千中不同的数据类型，HTTP在传输数据对象时会为他们打上称为MIME的数据格式标签，用于区分数据类型。最初MIME是用于电子邮件系统的，后来HTTP也采用了这一方案。

在HTTP协议消息头中，使用Content-Type来表示请求和响应中的媒体类型信息。它用来告诉服务端如何处理请求的数据，以及告诉客户端（一般是浏览器）如何解析响应的数据，比如显示图片，解析并展示html等等。

### Content-Type格式

Content-Type：type/subtype ;parameter

type：主类型，任意的字符串，如text，如果是\*号代表所有；

Text：用于标准化地表示的文本信息，文本消息可以是多种字符集和或者多种格式的；

Multipart：用于连接消息体的多个部分构成一个消息，这些部分可以是不同类型的数据；

Application：用于传输应用程序数据或者二进制数据；

Message：用于包装一个E-mail消息；

Image：用于传输静态图片数据；

Audio：用于传输音频或者音声数据；

Video：用于传输动态影像数据，可以是与音频编辑在一起的视频数据格式。

subtype：子类型，任意的字符串，如html，如果是\*号代表所有，用“/”与主类型隔开；

text/plain（纯文本）

text/html（HTML文档）

image/gif（GIF图像）

image/jpeg（JPEG图像）【PHP中为：image/pjpeg】

application/octet-stream（任意的二进制数据）

application/pdf（PDF文档）

application/x-www-form-urlencoded（使用HTTP的POST方法提交的表单）

multipart/form-data

parameter：可选参数，如charset，boundary等。

charset: 指定字符编码的标准

boundary:

### Content-Type常见的类型

### 表单提交方式

格式:Content-Type: application/x-www-form-urlencoded;charset=utf-8

提交数据格式:

1、数据格式按照key1=val1&key2=val2。

2、提交数据中有特殊字符和中文，会进行url编码。

3、Form表单中默认的为该格式。

最常见的 POST 提交数据的方式,浏览器的原生 form 表单，如果不设置 encType 属性

编码原理

step 1 ：将保留字符转化为对应的 ASCⅡ 字节值 （byte value）;

step 2 ：将对应的对应的 ASCⅡ 字节值（十进制）转换成对应的十六进制值。

step 3 ：在对应十六进制值前面加上百分号 % 。

将需要转码的字符转为16进制，然后从右到左，取4位(不足4位直接处理)，每2位做一位，前面加上%，编码成%XY格式。空格ASCII码是32，对应16进制是20，那么urlencode编码结果是:%20,但在新标准中空格对应的是+,见RFC-1738中ASCII码是-10544，对应的16进制是D6D0，那么urlencode编码结果是:%D6%D0。

为什么要使用 URL编码 百分号编码

每一个 URL 都是单独的字符串。作为一个连续的字符串，URL 是不能存在空格的。所以在使用 URL 的时候需要使用其他的字符代替空格。

URL 中经常使用到的 ？ 和 # 等符号在 URL 中具有特殊的定义，在正常使用时，不能直接使用这些符号。所以我们需要使用其他字符代替这些符号。

根据 RFC 3986 协议定义，URL 对于在 URL 中用到的特殊字符和关键字，都需要使用特定的 URL 编码进行转化。

### 表单提交(多类型数据提交)

格式: Content-Type: multipart/form-data; boundary=${bound}

Boundary: 分隔符,用来分割多个请求实体。

数据格式

|  |
| --- |
| Header (请求头)  <空行>(以下为请求实体部分)  --${ boundary} (boundary:分隔符)  Content-Disposition: form-data; name="<服务器端需要知道的名字>"; filename="<服务器端这个传上来的文件名>"  Content-Type: application/zip (实体的文件类型)  <空行>  <二进制数据 body>  --${ boundary}--(boundary:分隔符)  （上面那些都要转为二进制文件）最终都是二进制文件上传的 |

例如:

Postman上传文本,excel,图片格式内容

图片包含 游戏机, 截图

描述已自动生成

|  |
| --- |
| POST http://babel.confucius.mobi/file/upload/ HTTP/1.1  Content-Type: multipart/form-data; boundary=--------------------------105544843373902367487191  User-Agent: PostmanRuntime/7.26.1  Accept: \*/\*  Cache-Control: no-cache  Postman-Token: 52379f07-6b40-4d18-ba0b-ba41542557c1  Host: babel.confucius.mobi  Accept-Encoding: gzip, deflate, br  Connection: keep-alive  Content-Length: 38808  ----------------------------105544843373902367487191  Content-Disposition: form-data; name="key1"  val1  ----------------------------105544843373902367487191  Content-Disposition: form-data; name="image"; filename="test.jpg"  Content-Type: image/jpeg  image实体  ----------------------------105544843373902367487191  Content-Disposition: form-data; name="excel"; filename="test.xlsx"  Content-Type: application/vnd.openxmlformats-officedocument.spreadsheetml.sheet  excel实体  ----------------------------105544843373902367487191-- |

Java代码

|  |
| --- |
| **public static void** main(String[] args) **throws** IOException {  String url = **"http://babel.confucius.mobi/file/upload/"**;  URL url1 = **new** URL(url);  HttpURLConnection conn = (HttpURLConnection)url1.openConnection();  conn.setDoOutput(**true**);  conn.setDoInput(**true**);  conn.setRequestMethod(**"POST"**);  String boundary = **"-----------------------------"** + System.*currentTimeMillis*();  *//设置实体类型* conn.setRequestProperty(**"Content-Type"**, **"multipart/form-data; boundary="** + boundary);  *//设置响应体* OutputStream out = conn.getOutputStream();  *//分割线* out.write((**"--"**+boundary + **"\r\n"**).getBytes());*//换行  //实体信息* out.write(  String.*format*(**"Content-Disposition: form-data; name=\"file\"; filename=\"%s\"\r\n"**, **"file.jpg"**)  .getBytes());  *//实体类型* out.write(**"Content-Type: image/jpeg \r\n\r\n"**.getBytes());  *//实体byte* **byte**[] data = **new byte**[1024];  **int** len = 0;  FileInputStream input = **new** FileInputStream(**new** File(**"I:\\test.jpg"**));  **while** ((len = input.read(data)) > -1) {  out.write(data, 0, len);  }  *//分割线* out.write((**"\r\n--"** + boundary + **"--"**).getBytes());  out.close();  input.close();  InputStream resp = conn.getInputStream();  StringBuffer sb = **new** StringBuffer();  **while** ((len = resp.read(data)) > -1) {  sb.append(**new** String(data, 0, len, **"utf-8"**));  }  resp.close();  String result = sb.toString(); } |

### Json提交方式

格式: Content-Type: application/json;charset=utf-8

数据格式: {"key1": "val1","key2": "val2"}

### Content-Type值的类型

1 application/json：消息主体是序列化后的 JSON 字符串

2 application/x-www-form-urlencoded：数据被编码为名称/值对。这是标准的编码格式

3 multipart/form-data： 需要在表单中进行文件上传时，就需要使用该格式。常见的媒体格式是上传文件之时使用的

4 text/plain：数据以纯文本形式(text/json/xml/html)进行编码，其中不含任何控件或格式字符。

### Content-Type对get请求和post请求的影响

GET 请求不存在请求实体部分，键值对参数放置在 URL 尾部，浏览器把form数据转换成一个字串（name1=value1&name2=value2...），然后把这个字串追加到url后面，用?分割，加载这个新的url。因此请求头不需要设置 Content-Type 字段

非 ASCII 码会自动进行编码转换，例如发送请求：www.bilibili.com?hehe=你的我的

值得一提的是，GET 参数的编码方式是无法人为干涉的，这导致了不同浏览器有不同的编码方式，因此最稳妥的方案是人工预编码，人工解码，从而禁止浏览器编码的干涉

### POST 请求

Http Header里的Content-Type一般有这三种：

1.application/x-www-form-urlencoded：数据被编码为名称/值对。这是标准的编码格式。默认行为。会将表单内的数据转换拼接成 key-value 对（非 ASCII 码进行编码）

2.multipart/form-data(一般用来上传文件)： 数据被编码为一条消息，页上的每个控件对应消息中的一个部分

3.text/plain： 数据以纯文本形式(text/json/xml/html)进行编码，其中不含任何控件或格式字符。postman软件里标的是RAW。（中文不进行编码）

### AJAX请求时最好要通过设置请求头来指定为application/x-www-form-urlencoded编码类型

### 表单上传文件时必须指定编码类型为"multipart/form-data

## Content-Length

### Content-length简述

Content-length：文中实体主体的字节大小.

1、这个大小是包含了所有内容编码的, 比如, 对文本文件进行了gzip压缩的话, Content-Length首部指的就是压缩后的大小而不是原始大小.

2、使用十进制的数字表示了消息的长度。

3、实体内容编码后的字节数量为字节大小的值。

### Content-Length

content-Length首部指示出报文中实体主体的字节大小。这个大小是包含了所有内容编码的,比如,对文本文件进行了gzip压缩的话, content-Length首部就是压缩后的大小,而不是原始大小。

1、检测结尾

HTTP的早期版本采用关闭连接的办法来划定报文的结束。但是,没有Content-Length的话,客户端无法区分到底是报文结束时正常的连接关闭,还是报文传输[344中由于服务器崩溃而导致的连接关闭。客户端需要通过content-Length来检测报文截尾。

S

### 客户端获取消息长度

\* 响应为1xx，204，304相应或者head请求，则直接忽视掉消息实体内容。

\* 如果有Transfer-Encoding，则优先采用Transfer-Encoding里面的方法来找到对应的长度。比如说Chunked模式。

\* “如果head中有Content-Length，那么这个Content-Length既表示实体长度，又表示传输长度。如果实体长度和传输长度不相等(比如说设置了Transfer-Encoding)，那么则不能设置Content-Length。如果设置了Transfer-Encoding，那么Content-Length将被忽视”。这句话翻译的优点饶，其实关键就一点：有了Transfer-Encoding，则不能有Content-Length。

\* 通过服务器关闭连接能确定消息的传输长度。(请求端不能通过关闭连接来指明请求消息体的结束，因为这样可以让服务器没有机会继续给予响应)。这种情况主要对应为短连接，即非keep-alive模式。

HTTP1.1必须支持chunk模式。因为当不确定消息长度的时候，可以通过chunk机制来处理这种情况。

\* 在包含消息内容的header中，如果有content-length字段，那么该字段对应的值必须完全和消息主题里面的长度匹配。

######其实后面几条几乎可以忽视，简单总结后如下：

1、Content-Length如果存在并且有效的话，则必须和消息内容的传输长度完全一致。(经过测试，如果过短则会截断，过长则会导致超时。)

2、如果存在Transfer-Encoding(重点是chunked)，则在header中不能有Content-Length，有也会被忽视。

3、如果采用短连接，则直接可以通过服务器关闭连接来确定消息的传输长度。(这个很容易懂)

######结合HTTP协议其他的特点，比如说Http1.1之前的不支持keep alive。那么可以得出以下结论：

1、在Http 1.0及之前版本中，content-length字段可有可无。

2、在http1.1及之后版本。如果是keep alive，则content-length和chunk必然是二选一。若是非keep alive，则和http1.0一样。content-length可有可无。

## Contet-encoding

### 概述

Accept-Encoding 和 Content-Encoding 是 HTTP 中用来对「采用何种编码格式传输正文」进行协定的一对头部字段。它的工作原理是这样：浏览器发送请求时，通过 Accept-Encoding 带上自己支持的内容编码格式列表；服务端从中挑选一种用来对正文进行编码，并通过 Content-Encoding 响应头指明选定的格式；浏览器拿到响应正文后，依据 Content-Encoding 进行解压。当然，服务端也可以返回未压缩的正文，但这种情况不允许返回 Content-Encoding。这个过程就是 HTTP 的内容编码机制。

内容编码目的是优化传输内容大小，通俗地讲就是进行压缩。一般经过 gzip 压缩过的文本响应，只有原始大小的 1/4。对于文本类响应是否开启了内容压缩，是我们做性能优化时首先要检查的重要项目；而对于 JPG / PNG 这类本身已经高度压缩过的二进制文件，不推荐开启内容压缩，效果微乎其微还浪费 CPU。

### 流程

1、首先浏览器（也就是客户端）发送请求时，通过Accept-Encoding带上自己支持的内容编码格式列表；

2、服务端在接收到请求后，从中挑选出一种用来对响应信息进行编码，并通过Content-Encoding来说明服务端选定的编码信息

3、浏览器在拿到响应正文后，依据Content-Encoding进行解压。

### 三种数据压缩格式

DEFLATE，是一种使用 Lempel-Ziv 压缩算法（LZ77）和哈夫曼编码的数据压缩格式。定义于 RFC 1951 : DEFLATE Compressed Data Format Specification；

ZLIB，是一种使用 DEFLATE 的数据压缩格式。定义于 RFC 1950 : ZLIB Compressed Data Format Specification；

GZIP，是一种使用 DEFLATE 的文件格式。定义于 RFC 1952 : GZIP file format specification；

## http压缩

图片包含 笔记本, 游戏机, 控制, 电脑

描述已自动生成

### 背景

　　最近做了个项目，遇到这么一个问题：服务器返回给客户端的json数据量太大（大概65M），在客户端加载了1分多钟才渲染完毕(当然这加载时间也和本地的下行带宽有关)，费时耗流量，用户体验极其不好。后来网上搜优化的方法，就是Http压缩。

　　HTTP压缩可以大大提高浏览网站的速度，它的原理是，在客户端请求服务器对应资源后，从服务器端将资源文件压缩，再输出到客户端，由客户端的浏览器负责解压缩并浏览。即：通过减小HTTP响应大小来减少响应时间。相对于普通的浏览过程HTML ,CSS,Javascript , Text ，它可以节省40%左右的流量。更为重要的是，它可以对动态生成的，包括CGI、PHP , JSP , ASP , Servlet,SHTML等输出的网页也能进行压缩，压缩效率也很高。而GZIP本身就是一种网络流压缩算法，而且应用相当广泛。本文是针对apache tomcat 8.0.47进行配置GZIP压缩的。浏览器使用Mozilla Firefox 35.0.1，调试用自带的Firebug，以下和网络有关的截图来自Firebug控制台。

### 简述

5. 协议压缩就是依据HTTP协议进行压缩，不需要程序员进行压缩，解压编码，而是把压缩过程交给WEB服务器，将解压过程交给客户端。 如果客户端为支持GZIP压缩的浏览器，那么解压过程也不需要程序员参与，浏览器会按照一定的规则自动进行解压缩；如果客户端为HttpClient ，那么就需要手动进行GZIP解码了。

 6. 压缩过程：客户端发送http请求，如果请求头header中携带Accept-Encoding: gzip,deflate (现在的浏览器一般默认都是这样)，那么浏览器的意思是:服务器需要进行GZIP压缩,再看响应内容的类型是否满足服务器配置的需要压缩的类型，如果符合，那么WEB服务器在传输响应内容之前，会对响应内容进行压缩，并在响应头中添加Content-Encoding gzip；如果不符合，那么将不压缩，直接返回。

7. 解压过程：（浏览器）客户端接收到响应，如果响应头中包含Content-Encoding GZIP，那么浏览器会自动将响应内容进行GZIP解压缩，然后再呈现在页面上。如果不包含，那么将直接呈现在页面上。

### 压缩的优缺点

1. http压缩对纯文本可以压缩至原内容的40%, 从而节省了60%的数据传输。

### 压缩流程

手机屏幕截图

描述已自动生成

1. 浏览器发送Http request 给Web服务器, request 中有Accept-Encoding: gzip, deflate。 (告诉服务器， 浏览器支持gzip压缩)

2. Web服务器接到request后， 生成原始的Response, 其中有原始的Content-Type和Content-Length。

3. Web服务器通过Gzip，来对Response进行编码， 编码后header中有Content-Type和Content-Length(压缩后的大小)， 并且增加了Content-Encoding:gzip. 4. 把Response发送给浏览器。

5. 浏览器接到Response后，根据Content-Encoding:gzip来对Response 进行解码。 获取到原始response后， 然后显示出网页。

6、响应头 只有对实体内容进行了压缩,才会有content-type,否则会出现解码失败问题。

### Nginx配置压缩

### Tomcat配置压缩

|  |
| --- |
| <Connector port="8080"  protocol="HTTP/1.1"  connectionTimeout="20000"  redirectPort="8443"  compression="on"  compressionMinSize="2048"  noCompressionUserAgents="gozilla, traviata"  compressableMimeType="text/html,text/xml,text/javascript, application/javascript,text/css,text/plain,text/json"/> |

参数说明：

　　1、compression="on" 开启压缩。可选值："on"开启，"off"关闭，"force"任何情况都开启。

　　2、compressionMinSize="2048"大于2KB的文件才进行压缩。用于指定压缩的最小数据大小，单位B，默认2048B。注意此值的大小，如果配置不合理，产生的后果是小文件压缩后反而变大了，达不到预想的效果。

　　3、noCompressionUserAgents="gozilla, traviata"，对于这两种浏览器，不进行压缩（我也不知道这两种浏览器是啥，百度上没找到），其值为正则表达式，匹配的UA将不会被压缩，默认空。

4、compressableMimeType="text/html,text/xml,application/javascript,text/css,text/plain,text/json"会被压缩的MIME类型列表，多个逗号隔，表明支持html、xml、js、css、json等文件格式的压缩（plain为无格式的，但对于具体是什么，我比较概念模糊）。compressableMimeType很重要，它用来告知tomcat要对哪一种文件进行压缩，如果类型指定错误了，肯定是无法压缩的。那么，如何知道要压缩的文件类型呢？可以通过以下这种方法找到。

手机屏幕截图

描述已自动生成

### Fiddler流程

## Content-location

<https://cloud.tencent.com/developer/section/1189923>

### 简介

Content-Location报头指示为返回的数据的备用位置。主要用途是指示作为内容协商结果传输的资源的 URL 。

1. 该头用来存储要跳转的连接,例如,提交数据成功后，可以将一些下载等其他连接放到该字段中。

## Content-Language

### 简述

Content-Language [实体头](https://developer.mozilla.org/en-US/docs/Glossary/entity_header)用于描述旨在为观众的语言（S），使得其允许用户根据用户自己的优选语言区分。

# Ajax

## 什么是Ajax

### Ajax简介

Ajax(Asynchronous JavaScript and XML) 异步JavaScript和XML

从上面的解释中可以知道：ajax是一种技术方案，但并不是一种**新技术**。它依赖的是现有的CSS/HTML/Javascript，而其中最核心的依赖是浏览器提供的XMLHttpRequest对象，是这个对象使得浏览器可以发出HTTP请求与接收HTTP响应。

所以我用一句话来总结两者的关系：我们使用XMLHttpRequest对象来发送一个Ajax请求。

Ajax实际上是下面这几种技术的融合：

(1)XHTML和CSS的基于标准的表示技术

(2)DOM进行动态显示和交互

(3)XML和XSLT进行数据交换和处理

(4)XMLHttpRequest进行异步数据检索

(5)Javascript将以上技术融合在一起

客户端与服务器，可以在【不必刷新整个浏览器】的情况下，与服务器进行异步通讯的技术

### 为什么我们需要Ajax？

在我们之前的开发，每当用户向服务器发送请求，哪怕只是需要更新一点点的局部内容，服务器都会将整个页面进行刷新。

性能会有所降低(一点内容，刷新整个页面！)

用户的操作页面会中断(整个页面被刷新了)

Ajax就是能够做到局部刷新！

## XMLHttpRequest

### XMLHttpRequest简介

XMLHttpRequest对象是Ajax中最重要的一个对象。使用Ajax更多的是编写客户端代码，而不是服务端的代码。

### XmlHttpReques发展历程

XMLHttpRequest一开始只是微软浏览器提供的一个接口，后来各大浏览器纷纷效仿也提供了这个接口，再后来W3C对它进行了标准化，提出了[XMLHttpRequest标准](https://www.w3.org/TR/XMLHttpRequest/" \t "_blank)。XMLHttpRequest标准又分为Level 1和Level 2。  
XMLHttpRequest Level 1主要存在以下缺点：

受同源策略的限制，不能发送跨域请求；

不能发送二进制文件（如图片、视频、音频等），只能发送纯文本数据；

在发送和获取数据的过程中，无法实时获取进度信息，只能判断是否完成；

那么Level 2对Level 1 进行了改进，XMLHttpRequest Level 2中新增了以下功能：

可以发送跨域请求，在服务端允许的情况下；

支持发送和接收二进制数据；

新增formData对象，支持发送表单数据；

发送和获取数据时，可以获取进度信息；

可以设置请求的超时时间；

### XMLHttpRequest 工作原理

传统的web前端与后端的交互中，浏览器直接访问Tomcat的Servlet来获取数据。Servlet通过转发把数据发送给浏览器。

当我们使用AJAX之后，**浏览器是先把请求发送到XMLHttpRequest异步对象之中，异步对象对请求进行封装，然后再与发送给服务器。服务器并不是以转发的方式响应，而是以流的方式把数据返回给浏览器**

XMLHttpRequest异步对象会不停监听服务器状态的变化，得到服务器返回的数据，就写到浏览器上【因为不是转发的方式，所以是无刷新就能够获取服务器端的数据】

# 跨域



## 什么是跨域

### 什么是跨域问题

前端调用的后端接口不属于同一个域（域名或端口不同），就会产生跨域问题，也就是说你的应用访问了该应用域名或端口之外的域名或端口。

电脑萤幕的截图

描述已自动生成

### 同源策略

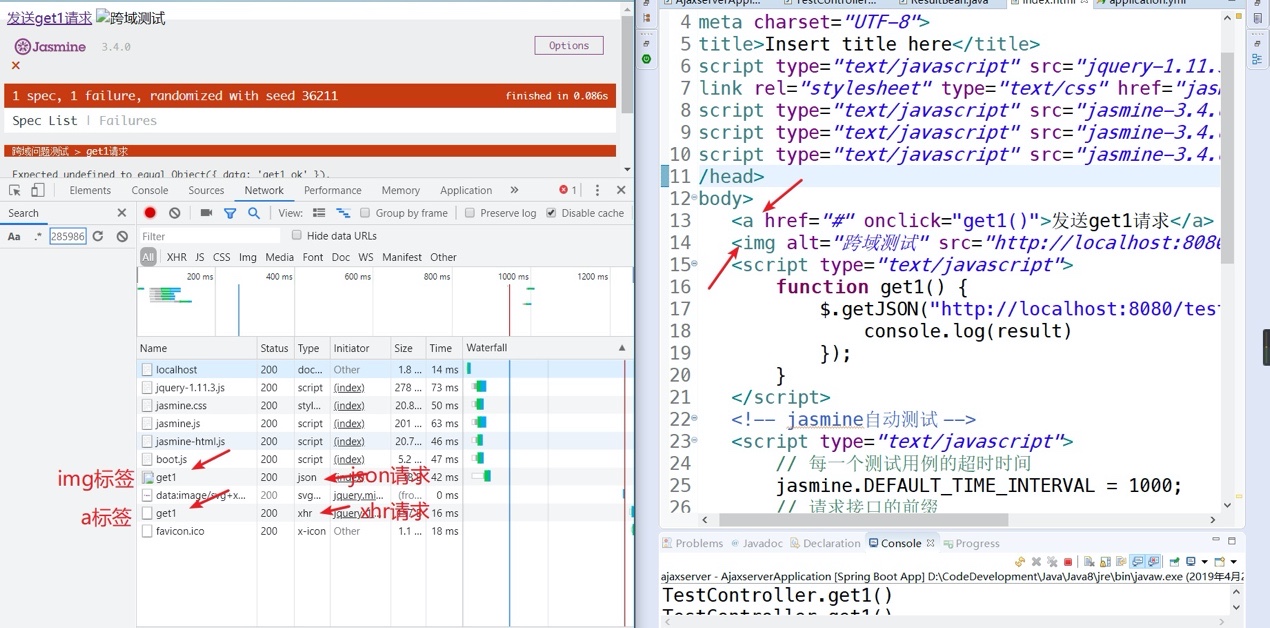
## 为什么会发生跨域问题

要同时满足三个条件才会产生跨域问题，这也就是为什么会产生跨域的原因。

1、浏览器限制，而不是服务端限制，可以查看Network，请求能够正确响应，response返回的值也是正确的

2、请求地址的域名或端口和当前访问的域名或端口不一样

3、发送的是XHR（XMLHttpRequest）请求，可以使用 a 标签（模拟xhr请求）和 img 标签（模拟json请求）做对比（控制台只报了一个跨域异常）



## 浏览器判断请求是否是跨域

### 浏览器判断请求是否跨域方法

浏览器会根据同源策略来判断一个请求是不是跨域请求。

非跨域请求：在请求头中会只包含请求的主机名：

社交网站的手机截图

描述已自动生成

跨域请求，在请求头中会既包含要请求的主机名还包括当前的源主机名，如果这两者不一致，那就是跨域请求了。

手机截图图社交软件的信息

描述已自动生成

### 浏览器对请求的分类

请求方法为 GET、POST、HEAD，请求头header中无自定义的请求头信息，请求类型Content-Type 为 text/plain、multipart/form-data、application/x-www-form-urlencoded 的请求都是简单请求。

**简单请求**：浏览器先发送（执行）请求然后再判断是否跨域。

请求方法为 GET、POST、HEAD，请求头header中无自定义的请求头信息，请求类型Content-Type 为 text/plain、multipart/form-data、application/x-www-form-urlencoded 的请求都是简单请求。

**非简单请求**：浏览器先发送预检命令（OPTIONS方法），检查通过后才发送真正的数据。

预检命令会发送自定义头为Access-Control-Request-Headers: content-type的请求到服务器，根据响应头的中的 “Access-Control-Allow-Headers”: “Content-Type” 判断服务器是否允许跨域访问。预检命令是可以缓存，服务器端设置 “Access-Control-Max-Age”: “3600”，这样后面发送同样的跨域请求就不需要先发送预检命令了。

请求方法为 PUT、DELETE 的 AJAX 请求、发送 JSON 格式的 AJAX 请求、带自定义头的 AJAX 请求都是非简单请求。

## 解决跨域问题的三种思路

1、客户端浏览器解除跨域限制（理论上可以但是不现实）

2、发送JSONP请求替代XHR请求（并不能适用所有的请求方式，不推荐）

3、修改服务器端（包括HTTP服务器和应用服务器）（推荐）

### 2.1 客户端浏览器解除跨域限制

浏览器默认都是开启跨域安全检查的，我们可以使用命令行启动浏览器，加上禁止安全检查的参数，以谷歌浏览器为例，chrome.exe --disable-web-security --user-data-dir=E:/temp --user-data-dir 为浏览器缓存临时目录，浏览器这时会提示安全问题。

## Jsonp请求

### JSONP 是什么

JSONP（JSON with Padding）是JSON的一种补充使用方式，不是官方协议，而是利用 Script 标签请求资源可以跨域的特点，来解决跨域问题的，是一种变通的解决方案。

### 使用 JSONP，服务器后台需要改动吗

答案是需要，这里以Spring Boot为例，在 Spring Boot 1.5 大版本中，添加一个切面来支持JSONP请求，注意在 Spring Boot 2.x 大版本中已经废弃了 AbstractJsonpResponseBodyAdvice 类，不推荐这种方式解决跨域问题。

AJAX代码如下：

|  |
| --- |
| $.ajax({  url: baseUrl + "/get1",  dataType: "jsonp", // 关键字段  jsonp: "callback", // 前后端默认的约定  cache: true, // 表示请求结果可以被缓存，url中不会有下划线参数了  success: function(json) {  result = json;  }  }); |

服务端代码：

|  |
| --- |
| @ControllerAdvice  public class JsonpAdvice extends AbstractJsonpResponseBodyAdvice {  public JsonpAdvice() {  super("callback");  }  } |

### JSONP 实现原理

JSONP请求的类型是JavaScript脚本（callback 作为前后端的约定，callback的值做为方法名，json内容作为方法的参数），而XHR请求的类型是json类型。

电脑屏幕的截图

描述已自动生成

可以在浏览器中查看 Jquery 源码来验证 JSONP 是否将请求包装成了 script 脚本。

社交网站的手机截图

描述已自动生成

在Jquery 源码中打断点。

手机截图图社交软件的信息

描述已自动生成

刷新后查看 element 元素，可以看到 Jquery 在 html 源码中添加了 script 标签。

手机截图图社交软件的信息

描述已自动生成

## JSONP 的缺点

1. 只支持 GET 方法请求，不管 AJAX 中实际的请求方法是不是 GET

2、服务端还需要修改代码

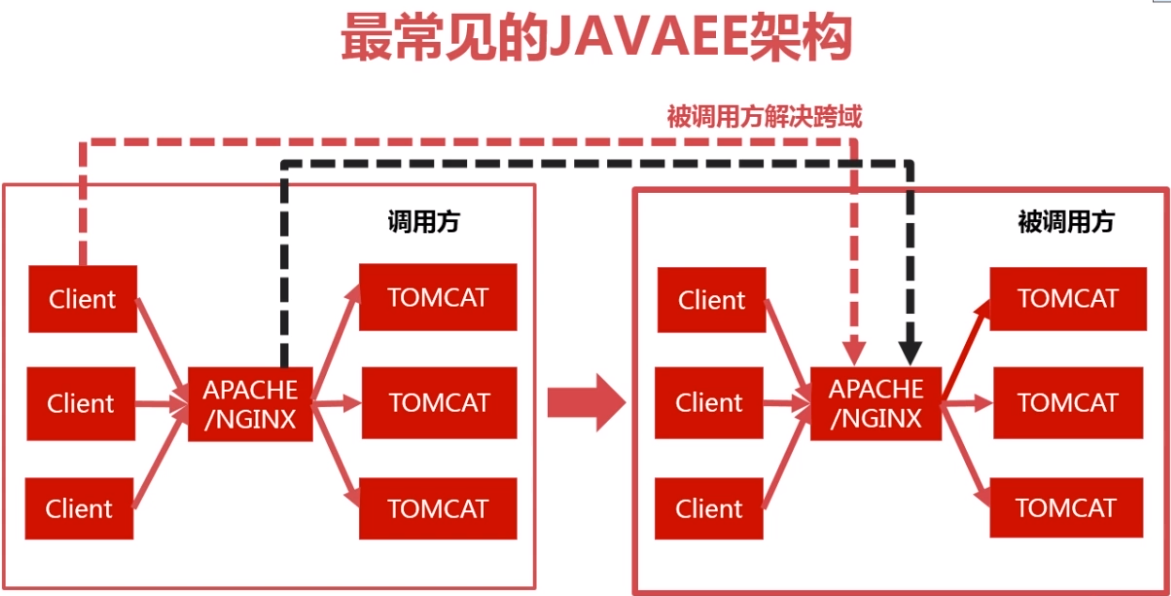
3、发送的不是 XHR 请求，无法使用 XHR 对象（但这也是为什么可以解决跨域问题的根本）

总之，并不推荐使用 JSONP 方式来解决跨域问题，因为还有更好的解决方式。

## 修改服务器端

最核心还是因为HTTP无状态，要有cookies配合进行用户标识，而浏览器储存了cookies之后，浏览器会在访问URL时自动带上cookies，服务器端根据cookies来认证你是谁，有什么权限可以做什么事情。关键就是这个。跨域的限制不是因为服务器端，服务器端只能接收是谁发起了请求并不能知道从哪个域名下发起的请求，所以跨域的限制来自于浏览器，浏览器拦截了服务器端发回的请求，浏览器要求服务器端出示哪些域名可以获取这个响应的资源。

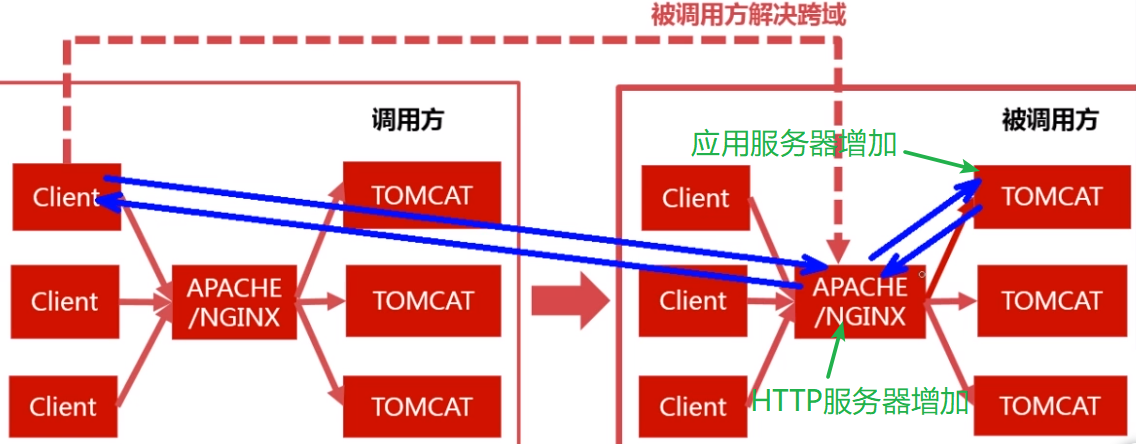
根据现如今网站架构设计，可以将前端应用看作调用方使用服务，将后端应用看作被调用方提供服务。



根据服务器的作用，可以将服务器分为 HTTP 服务器和应用服务器，所有修改服务器端既可以是修改应用服务器，也可以是修改 HTTP 服务器。

### 被调用方修改

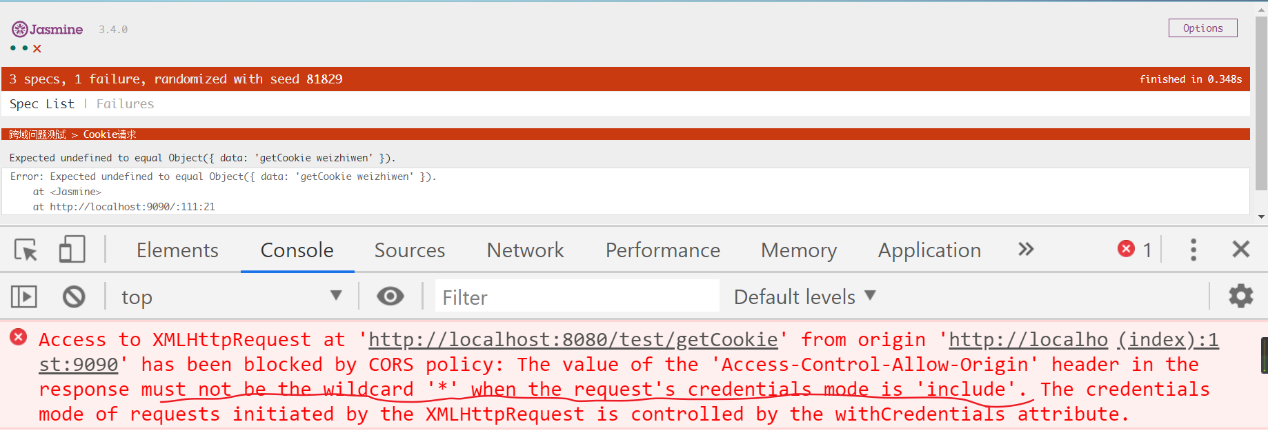
被调用方的解决思路是在响应头中增加指定的字段允许调用方服务器跨域调用。



在应用服务器增加指定字段对于不带 Cookie 的跨域请求，设置允许跨域的原始域名为任意域名，

“Access-Control-Allow-Origin”: “\*”，设置允许跨域的方法为任意方法，

“Access-Control-Allow-Methods”: “\*”，但是这样的星号设置不能满足带 Cookie 的跨域请求。



带自定义头的跨域请求，设置允许跨域的请求头自定义的请求头，“Access-Control-Allow-Headers”:“自定义的请求头”

在 Java Web 中，可以添加一个过滤器来设置上面的参数。



而使用 Spring Boot 框架，只需要在 Controller 类上加上 @CrossOrigin 注解就可以轻松解决跨域问题了。

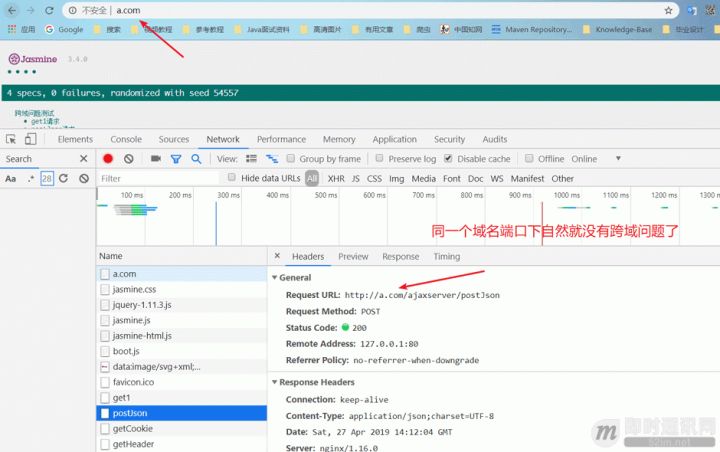
### 在 HTTP 服务器增加指定字段

以常用的 Nginx 服务器和 Apache 服务器为例。

Nginx 服务器允许跨域配置（注意不要手动直接点击Nginx.exe，否则停止和重新载入配置会失败的）：

## 调用方修改

调用方的解决思路是反向代理，也即是将被调用方的域名代理到调用方域名下，这样就符合同源策略了，也就解决了跨域问题。



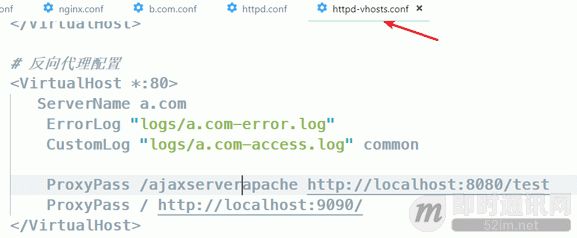
调用方反向代理效果演示

调用方修改一般都是直接修改 HTTP 服务器配置。

Nginx 服务器反向代理配置



Apache 服务器反向代理配置：



# 国际化